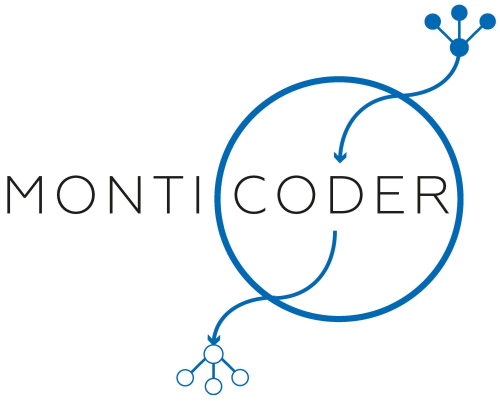
SE | RWTH AACHEN UNIVERSITY

Lars Hermerschmidt

Agile Modellgetriebene Entwicklung von Software Security & Privacy



Aachener Informatik-Berichte,
Software Engineering

Band 41

Hrsg: Prof. Dr. rer. nat. Bernhard Rumpe

Agile Modellgetriebene Entwicklung von Software Security & Privacy

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Lars Hermerschmidt
aus Burgwedel

Berichter: Universitätsprofessor Dr. rer. nat. Bernhard Rumpe
Universitätsprofessor Dr. Eric Bodden

Tag der mündlichen Prüfung: 29.10.2018

[Her19] L. Hermerschmidt:
Agile Modellgetriebene Entwicklung von Software Security & Privacy.
Shaker Verlag, ISBN 978-3-8440-6707-1. Aachener Informatik-Berichte, Software Engineering, Band 41. Juni 2019.
www.se-rwth.de/publications/

Kurzfassung

Durch die zunehmende Verbreitung von IT Systemen in jedem Bereich unseres Lebens wachsen die Anforderungen an die korrekte Funktionsweise dieser Systeme. Außerdem werden immer mehr Informationen in diesen Systemen gespeichert mit dem Ziel das Leben angenehmer und effizienter zu machen. Gleichzeitig werden täglich im Durchschnitt 16 neue Vulnerabilities in Softwareprodukten veröffentlicht, die es Angreifern ermöglichen Funktionen auszuführen, die von den Entwicklern dieser Produkte nicht vorgesehen waren [Nat15]. Die Sorge von Nutzern, dass unliebsame Dritte Zugriff auf ihre Daten besitzen besteht jedoch nicht nur, wenn ein System durch einen Angreifer kompromittiert wird, sondern auch bei Services, die Nutzern nicht transparent darlegen zu welchem Zweck ihre Daten erhoben und verarbeitet werden.

Diese Security- und Privacy-Aspekte müssen in der Entwicklung von Software Systemen adressiert werden. Allerdings besitzen Entwickler häufig eine rein konstruktive Sichtweise auf ein System, sodass sie sich nur schwer in die Lage eines Angreifers versetzen können. Daher werden in dieser Arbeit die drei Aspekte Security-Architektur, die korrekte Verarbeitung von Eingabedaten bei der Erstellung von Ausgabedaten und die Privacy von Nutzern im Kontext der Softwareentwicklung untersucht. Dabei werden Modelle und der agile modellgetriebene Softwareentwicklungsansatz eingesetzt, um Security- und Privacy-Aspekte explizit zu modellieren, analysieren und in ausführbare Systeme zu übersetzen.

Um die Security-Architektur eines Systems aus dem konstruktiven Blickwinkeln von Entwicklern zu beschreiben wird die Security-Architekturmodellierungssprache MontiSecArc vorgestellt. Aus dieser wird eine domänenspezifische Transformationssprache abgeleitet, die es erlaubt bekannte Vulnerabilities in der Security-Architektur zusammen mit einer Lösung in einem Flaw Correction Pattern zu beschreiben. Diese Pattern lassen sich automatisiert in einem agilen Entwicklungsprozess anwenden, um die beschriebenen Vulnerabilities zu erkennen und zu beheben. Außerdem erlauben es Flaw Correction Pattern bekannte Arten von Vulnerabilities präzise zu beschreiben, sodass diese benannt, unterschieden und gesammelt werden können.

Injection Vulnerabilities wie Cross-Site-Scripting (XSS) und SQL Injection entstehen, wenn Programme Eingabedaten verwenden, um Ausgaben in einer Sprache zu generieren, ohne dabei die Eingabedaten im Kontext der Ausgabesprache zu encoden. Dieses Encoding, welches Injection Vulnerabilities verhindert, ist spezifisch für jede Ausgabesprache und einzelne Kontexte innerhalb dieser Sprache. Daher wird in dieser Arbeit mit MontiCoder ein Ansatz vorgestellt, bei dem das kontextspezifische Encoding in die Definition der Sprache integriert wird und daraus Unparser und Parser generiert werden, die kontextspezifisches Encoding und Decoding automatisch durchführen. Dieser Ansatz erlaubt es Injection Vulnerabilities bereits bei der Definition einer Sprache zu verhindern, sodass diese bei der Nutzung der Sprache nicht entstehen.

Um Nutzern eines Cloud Services mehr Transparenz und Entscheidungshoheit über die Verwendung ihrer Daten zu ermöglichen wird in dieser Arbeit die Privacy Development Language (PDL) vorgestellt. Entwickler können diese zur Beschreibung der Datenstruktur eines Services nutzen, um den Nutzern des Services die Nutzung ihrer Daten transparent darzustellen. Dazu wird aus einem PDL-Modell eine interaktive Privacy Policy erzeugt. Diese erlaubt es Nutzern nur Teile eines Services zu verwenden und stellt gleichzeitig dar welche Daten dabei verarbeitet werden, sodass die Nutzer eine informierte Entscheidung über ihre Privacy treffen können.

Abstract

IT systems continue to penetrate all aspects of daily life, so that these systems are required to function correctly while they aggregate more and more information to enable a more comfortable and efficient life. However, every day 16 new vulnerabilities in average are reported in software products, which enable adversaries to abuse system functionality in developer unintended way [Nat15]. Nevertheless users do not only fear unauthorized access to their data by adversarial third party when they compromise system's security, but as well if services do not provide transparency over their collection and processing of sensitive user data. These security and privacy aspects need to be addressed during software system development, where developers typically focus on a constructive view on the system, which makes it hard for them to think like an adversary who misuses the system. In this thesis the three aspects security architecture, correct handling of input data during the creation of output data, and user privacy are researched in the context of software development. Models are used to describe these aspects and the agile model-driven software development approach is utilized to connect those models to executable systems.

To enable developers to express their constructive view on the security architecture of a system, the security architecture modeling language MontiSecArc is presented From this language a domain specific transformation language is derived, which enables security experts to express known flaws in the security architecture together with a solution in a flaw correction pattern. These pattern can be applied automatically within an agile development process to prevent architectural flaws. In addition the notation of flaw correction patterns enables precise description, naming, and distinction of architectural flaws such that flaw correction patterns can be collected.

Injection vulnerabilities like Cross-Site-Scripting (XSS) and SQL Injection are the most common class of vulnerabilities, which emerge if a program uses input data to create output data that is written in a language without encoding the input according to the output's language. This injection vulnerability preventing encoding is depending on the output language and the different contexts within the language. Therefore, in this work the MontiCoder approach is presented, which integrates the context-specific encoding into the language definition and derives unparser and parser from this definition, which automatically perform context-specific encoding and decoding. This approach shifts the definition of the correct encoding of data from the developer who is using a language to the language developer and provides the latter one the ability to precisely define the encoding together with the language itself.

Facing the challenges of developers who aim to provide more transparency and self-determinism to service users about the usage of there data within a service, in this thesis the Privacy Development Language (PDL) is presented. The PDL enables developers to model the service's data structure along with the usage of the data within the service. From this data model an interactive privacy policy is generated, which enables users to select those parts of the service they want to use and present them a description of the used data and the procession which is performed to deliver the selected service. This way the approach enables service users to perform an informed privacy decision and eases developers work to provide transparency.

Explicitly modeling these aspects enables analyses of security an privacy on model level, such that vulnerabilities are fixed within models as well. This way there is no need to extract all flaws from handwritten low level code after development.

Danksagung

An dieser Stelle möchte ich mich bei all den Menschen bedanken, die mich während meiner Promotion begleitet und unterstützt haben. Mein erster Dank gilt Prof. Dr. Bernhard Rumpe für das mir entgegengebrachte Vertrauen und die gewährten Freiheiten während des Entstehens dieser Arbeit. Ich danke ihm dafür mir die Möglichkeit der Promotion am Lehrstuhl für Software Engineering der RWTH Aachen gegeben zu haben besonders mit meiner Vertiefung im Bereich Security. Des Weitern möchte ich Prof. Dr. Eric Bodden für das Interesse an dieser Arbeit und die Bereitschaft zur Übernahme der Zweitkorrektur danken. Auch Prof. Dr. Klaus Wehrle möchte ich für die Leitung sowie Dr. Walter Unger für die Mitarbeit in meinem Prüfungskomitee danken.

Mein besonders großer Dank geht an meine Frau, die mich zur Promotion ermutigt und über die gesamte Zeit unterstützt hat. Auch meinen Kollegen am Lehrstuhl möchte ich meinen Dank aussprechen. Es war mir eine Freude mit euch spannende Forschungsfragen zu diskutieren und viel über Software Engineering zu lernen. Auch die Akquise von neuen Projekten, haben wir immer gemeinsam geschafft. Dafür möchte ich mich bei Vincent Bertram, Marita Breuer, Robert Eikermann, Timo Greifenberg, Dr. Arne Haber, Dr. Christoph Herrmann, Andreas Horst, Dr. Katrin Hölldobler, Thomas Kurpick, Achim Lindt, Dr. Markus Look, Dr. Klaus Müller, Dr. Pedram Mir Seyed Nazari, Antonio Navarro Perez, Dr. Claas Pinkernell, Dr. Dimitri Plotnikov, Deni Raco, Dr. Dirk Reiss, Dr. Jan Oliver Ringert, Dr. Alexander Roth, Dr. Martin Schindler, Steffi Schrader, Christoph Schulze, Galina Volkova, Michael von Wenckstern und Dr. Andreas Wortmann bedanken. Euch möchte ich außerdem für das wertvolle Feedback zu dieser Arbeit danken.

Ebenfalls hervorheben möchte ich die vielen Studenten, deren Abschlussarbeiten ich betreuen durfte. Euch bei der Erarbeitung neuen Wissens zu begleiten und zu unterstützen hat mir viel Freude gemacht. Ich hoffe euer Erkenntnisgewinn war mindestens genauso groß wie meiner.

Großen Dank möchte ich auch meinen Eltern aussprechen, die mich auf dem langen Weg bis zur Promotion fortwährend unterstützt haben. Es war sicherlich nicht immer einfach mit dem Informatiker in mir.

So long, and thanks for all the fish.

Hürth, April 2019
Lars Hermerschmidt

Inhaltsverzeichnis

Kapitel 1.
Einleitung

> Any jackass can kick down a barn, but it takes a good carpenter to build one.
>
> Sam Rayburn

Anfangs wurde Software auf Servern eingesetzt, um Geschäftsvorgänge wie Buchhaltung und Produktionsplanung zu optimieren. Heutzutage nutzen wir Software bereits in einem Großteil unseres täglichen Lebens. Und durch die zunehmende Verbreitung von Cyber Physical Systems (CPS) [Lee08], Internet of Things (IoT) [AIM10], Smartphones und Wareables wird die Digitalisierung unseres Lebens weiter voranschreiten. So durchdringt Software die privaten Bereiche unseres Lebens und verbindet sie mit Servern in der Cloud [AFG+10], die immer ausreichend verfügbare Ressourcen zur Speicherung von Daten und zur Berechnung von komplexen Algorithmen bereitstellt, um unser tägliches Leben zu verbessern. Um in den Genuss dieser Verbesserungen zu kommen, müssen wir uns darauf verlassen, dass Software und die Systeme die sie steuert fehlerfrei funktionieren. Somit kann ein Angreifer allein durch das Ausnutzen eines Fehlers in der Software, einer sogenannten Vulnerability, ein System absichtlich manipulieren und somit die Nutzer dieser Systeme in den Lebensbereichen kontrollieren und beeinflussen, in denen sie das System verwenden.

Die durch Edward Snowden veröffentlichten Informationen zeigen, dass zumindest bei Geheimdiensten ein großes Interesse daran besteht Menschen durch die Manipulation von digitalen Systemen zu überwachen und zu beeinflussen [Gid13]. Ein Bericht des Geheimdienstdirektors der Vereinigten Staaten [Cla16] nennt explizit die Vulnerabilities von Software, die in Geräten des Internet of Things eingesetzt wird als aussichtsreiche Möglichkeit zur Identifikation, Überwachung, Beobachtung und Lokalisierung von Personen. Weitere Ziele, die der Bericht benennt sind die Identifikation von Personen, die für eine bestimmte Aufgabe angeworben werden sollen, das Eindringen in Netzwerke, sowie die Erlangung von Zugriffsrechten von Personen.

Dabei ist das Internet of Things nur die neueste Technologie in der Software Vulnerabilities identifiziert wurden. In den vergangenen 10 Jahren wurden täglich ca. 16 Vulnerabilities in

kommerziell vertriebenen Software Produkten gemeldet, die bereits im Einsatz bei Kunden waren [Nat15][1]. Somit können nicht nur Geheimdienste, sondern auch das organisierte Verbrechen und Terroristen täglich neue Vulnerabilities nutzen, um ihre Ziele zu erreichen.

Allein durch diese technische Möglichkeit einer Überwachung wird der sogenannte Chilling Effekt beim Menschen ausgelöst [WZ75]. Dabei passen Menschen die befürchten überwacht zu werden, ihr Verhalten und ihre Meinungsäußerung an, sobald eine aufgezeichnete, von der Norm abweichende Meinung ihrer Einschätzung nach zu Repressalien führt. Um den Chilling Effekt in unserem digitalisierten Leben zu verhindern und so die weitere Digitalisierung privater Lebensbereiche zu ermöglichen, ohne dass dieser Eingriff zu einer Beeinflussung der Gesellschaft hin zu mehr sozialer Angepasstheit führt ist es notwendig Software Vulnerabilities zu eliminieren und Nutzern transparent zu machen welche persönlichen Informationen bei der Nutzung von Cyber Physical Systems an Cloud-Services übermittelt und zu welchem Zweck diese genutzt werden.

1.1. Kontext

Vulnerabilities sind Fehler die während der Softwareentwicklung entstehen, wenn beim Entwurf oder der Implementierung Sonderfälle im Programmablauf nicht hinreichend behandelt werden. Ein Angreifer kann diese Sonderfälle zu seinem Vorteil verwenden und so die Vulnerability ausnutzen. Vulnerabilities bleiben genauso wie andere Fehler, die in der Softwareentwicklung entstehen, häufig lange Zeit unbemerkt, wodurch die Kosten für ihre Beseitigung steigen [BB01] und auch das Risiko einer Ausnutzung besteht. Um sowohl dieses Risiko zu mindern, als auch die Kosten für die Fehlerbehebung zu minimieren, müssen Vulnerabilities bereits während der Entwicklung der Software behoben werden [McG03, McG06].

Im Entwurf werden die grundlegenden Entscheidungen über ein System getroffen und in der Architekturbeschreibung festgehalten. Da Fehler in der Softwareentwicklung annähernd zu gleichen Teilen im Entwurf und in der Implementierung entstehen [Gal04] und sich die Konsequenzen eines Fehlers aus dem Entwurf, einem sogenannten Flaw, in mehreren Teilen der Implementierung auswirkt ist es zur Reduktion der Fehlerbehebungskosten sinnvoll Flaws bereits während des Entwurfs in einer Security-Architekturanalyse zu identifizieren und zu beheben. In einem klassischen Entwicklungsprozess wird die Architekturbeschreibung von Entwicklern genutzt, um die Software in einer Programmiersprache zu implementieren. Dabei müssen sie technische Details hinzufügen, von denen im Entwurf abstrahiert wurde, um die grundlegenden Entscheidungen klarer darstellen zu können.

Ziel eines Softwareentwicklungsprojekts ist es eine Software unter gegebenen finanziellen und zeitlichen Bedingungen zu entwickeln, die den Vorstellungen von Funktion und Qualität des Auftraggebers entspricht. Es hat sich gezeigt, dass bei einem strikt sequenziellen Vorgehen, wie beispielsweise beim Wasserfallmodell, bei dem alle Anforderungen vor Beginn des Entwurfs und der Implementierung bekannt sein müssen die entwickelte Software in vielen Fällen nicht die Vorstellungen des Auftraggebers trifft [Boe88]. Als Reaktion darauf werden agile Vorgehensweisen angewandt, bei denen immer nur kleine Teile eines Systems entworfen, implementiert

[1] Dabei werden Vulnerabilities in Cloud Services und Individualsoftware wie Webapplikationen nicht mitgezählt.

und an den Kunden ausgeliefert werden. Dieses Vorgehen ermöglicht es dem Kunden frühzeitig zu beschreiben, welche Funktionen und Qualitätsmerkmale der Software noch hinzugefügt werden sollen [Bec00, BBvB+01]. In einem solchen agilen Entwicklungsprozess besteht die Herausforderung die Software möglichst leicht an sich ändernde Anforderungen anpassen zu können. Erfordert ein Änderungswunsch des Auftraggebers eine Änderung an der Architektur ist der leichteste Weg die Implementierung direkt, ohne Anpassung der Architekturbeschreibung, zu verändern. Durch dieses Vorgehen verliert die Architekturbeschreibung jedoch ihren Nutzen als Dokumentation, sodass die zukünftige Weiterentwicklung schwieriger wird. Um dies zu verhindern werden in der agilen modellgetriebenen Entwicklung Modelle, wie die Architekturbeschreibung, mit Hilfe von Generatoren automatisiert in eine Implementierung übersetzt [Rum11]. Entwickler implementieren dabei nicht mehr eine konkrete Implementierung, sondern nutzen oder erweitern Generatoren, die das Modell in eine Implementierung übersetzten und dabei technische Details hinzufügt. Dem Generat, welches einen Großteil der Implementierung ausmacht, wird bei Bedarf handgeschriebener Programmcode hinzugefügt der systemspezifische, technische Details implementiert, die nicht aus dem Modell abgeleitet werden.

Der Einsatz von Modellen bietet in der agilen modellgetriebenen Entwicklung die Möglichkeit Aspekte wie Security und Privacy, die in Programmiersprachen nicht direkt ausgedrückt werden, explizit im Entwurf zu modellieren und diese so in die Software einzubauen. Durch die automatische Übersetzung in die Implementierung lassen sich Aussagen über Flaws in den Modellen direkt auf das fertige Softwareprodukt übertragen und ihre Behebung ebenfalls in den Modellen durchführen, wie in dieser Arbeit dargestellt wird.

Damit Menschen ein System in ihrem Alltag nutzen und akzeptieren ist ein angemessenes Maß an Security notwendig, aber nicht ausreichend. Denn Nutzer wollen Systeme selbstbestimmt nutzen, wozu auch gehört, dass sie selbst darüber entscheiden können wollen zu welchem Zweck die über sie erhobenen Daten verarbeitet werden [Pea09]. Diesen Bedarf nach Privacy bewertet jeder Nutzer individuell unterschiedlich anhand der Anreize die ihm für die Preisgabe seiner Daten gegeben werden [Bec03]. Diese Anreize können sich über die Zeit und manchmal auch sehr spontan, je nach Lebenssituation ändern. So werden Nutzer eines Systems, welches kontinuierlich Vital- und Positionsdaten seiner Nutzer aufzeichnet darauf bedacht sein, dass diese Daten nicht öffentlich werden, da sich daraus sehr gut auf ihre Aktivitäten schließen lässt und so beispielsweise ein Arbeitgeber überprüfen kann, ob sein Arbeitnehmer wirklich krank im Bett liegt oder letzte Nacht zu lange gefeiert hat. In einer Notsituation würde ein Nutzer diese Daten allerdings freizügig preisgeben, wenn dies sein Leben rettet. Da Software und insbesondere Cloud Services jedoch nicht Nutzer-interindividuell gefertigt werden sondern von einem möglichst großen Nutzerkreis verwendet werden [AFG+10], muss die Software bereits während des Entwurfs für solche Situationen konzipiert werden und eine individuelle Konfiguration durch den Nutzer zulassen.

Security Vulnerabilities, die nicht im Entwurf, sondern in der Implementierung entstehen, sogenannte Bugs, sind oftmals eng mit der verwendeten Programmiersprache und den genutzten Frameworks verbunden, die Entwicklern nicht die richtige Funktionalität bieten, um Software ohne Vulnerabilities zu schreiben. Eine Lösung dieses Problems besteht darin die Entwickler im Umgang mit Programmiersprachen und Frameworks zu schulen, damit sie weniger Fehler bei deren Nutzung machen. Das Open Web Application Security Project (OWASP) stellt beispielsweise

Hinweise für Entwickler bereit, die beschreiben, wie bekannte Vulnerabilities in Webapplikationen vermieden werden können [Ope15b, WO13]. Ein sehr weit verbreiteter Bug, der unabhängig von der verwendeten Programmiersprache auftritt und doch nicht zu den Flaws gehört, ist eine Injection Vulnerability. Dabei gibt ein Programm A (beispielsweise ein Webserver) als Reaktion auf eine Eingabe ein Dokument in einer bestimmten Sprache aus, welche zur Kommunikation mit einem anderen Programm B (beispielsweise einem Web Browser) verwendet wird. Eine Injection Vulnerability liegt vor, wenn bei der Erstellung des Dokuments nicht überprüft wird, ob Eingabedaten, die zur Erstellung des Dokuments verwendet werden das Ausgabedokument in seiner Struktur so verändern, dass die Eingabedaten von B nicht als reine Daten, sondern als andere Teile des Dokuments verarbeitet werden. Ein Angreifer kann diese Vulnerability ausnutzen, um Dokumente im Namen von A an B zu schicken und so je nach Art des Dokuments Aktionen in B auszulösen, die vom Entwickler nicht vorgesehen waren.

1.2. Bestehende Ansätze und Verwandte Arbeiten

Für die zuvor genannten Vulnerabilities existieren zum Teil bereits Ansätze diese zu erkennen. Im Folgenden werden diese Ansätze zusammengefasst und ihre Limitierungen im Bezug auf den Einsatz in einem agilen modellgetriebenen Entwicklungsprozess dargestellt.

Für die Identifikation von Flaws ist Security-Expertenwissen notwendig, welches Entwickler in einem agilen Entwicklungsteam die an der Realisierung neuer Funktionen interessiert sind nicht mitbringen [Ale03]. Penetrationstests, ein etabliertes Verfahren bei dem ein System am Ende der Entwicklung von einem Security-Experten der in die Rolle eines Angreifers schlüpft mit Blackbox Tests untersucht wird, liefern aufgrund der fehlenden Information über Entwurf und Implementierung nur selten Flaws und geben vielmehr einen Eindruck dafür, wie schwer es Angreifer haben ein System zu kompromittieren. Existierende Methoden zur Identifikation von Flaws wie die Architectural Risk Analysis [McG06] setzen auf eine manuelle Analyse der Architektur durch Security-Experten. Zur Architekturbeschreibung existieren verschiedene Architekturbeschreibungssprachen (ADLs) [MT00], die sich jedoch auf die Beschreibung von Zugriffskontrolle und Verschlüsselung [RTDR05, RT06] oder verschiedene Security Level von Komponenten [MQRG97, ZAF08, HFM08, HWF+10] beschränken und nicht deren Zusammenhang darstellen.

Andere Ansätze wie UMLsec [Jür02], CORAS [SS13] und die Cyber Security Modeling Language [SEH13] verwenden in der Architekturbeschreibung Security-Fachtermini und modellieren den Angreifer, sodass Entwickler die kein Wissen über den Angreifer und die entsprechenden Termini haben das Architekturmodell nur schwer ohne einen Security-Experten erstellen können. Der bei Microsoft eingesetzte Threat Modeling Ansatz [HL06, S. 103], [Sho14] vermeidet dies trotz des anders lautenden Namen, sodass Entwickler selbständig Data Flow Diagrams (DFDs) erstellen können, die hierbei als Strukturbeschreibung genutzt werden. Zur Identifikation von Flaws wird die STRIDE Methode [Sho14, S. 61] auf ein DFD angewandt, die sechs Suchmuster einsetzt um Flaws zu identifizieren. Die Vielzahl an Security-Designprinzipien [SS75], Security Design Pattern [SFBH+13] und Security Design Flaws [IAD+14] lässt vermuten, dass weitere Suchmuster existieren, mit deren Hilfe Flaws automatisiert identifiziert werden können.

Zustandslose Kommunikationsverbindungen zwischen Clients und einem Server wie beispielsweise HTTP [FRe14a, FRe14b] werden eingesetzt damit der Server viele Anfragen einfach und schnell bearbeiten kann. Da Webanwendungen, die gezwungenermaßen HTTP verwenden, jedoch häufig Prozesse realisieren, die mehrere Zustände besitzen, die nur in einer bestimmten Reihenfolge durchlaufen werden dürfen muss die Anwendung sicherstellen, dass ein Angreifer einzelne Prozessschritte nicht überspringen kann. In einem modellgetriebenen Entwicklungsansatz besteht die Möglichkeit die erlaubten Zustände und Transitionen in einem Statechart zu modellieren und somit die Webanwendung gegen dieses spezifizierte Verhalten zu prüfen [BKK11].

Damit die Webapplikation genau die im Modell spezifizierten Transitionen zulässt übersetzt ein Generator das Statechart in eine Implementierung, die das Sessionmanagement realisiert. Um Vulnerabilities in einer solchen Implementierung zu identifizieren wird getestet ob, die im Modell nicht erlaubten Transitionen in der Implementierung durchgeführt werden können [BOS13]. Allerdings nennt der OWASP Testing Guide noch weitere Session-Management Vulnerabilities, die sich mit Hilfe eines funktionalen Modells identifizieren lassen [Ope15c].

Damit Informationen zwischen Programmen ausgetauscht werden können, müssen diese Programme Nachrichten in *einer* Sprache parsen (lesen) und unparsen (schreiben) können. Werden hierbei Sprachen verwendet, die in ihrer Komplexität eine kontextfreie Sprache übersteigen, lassen sich die Nachrichten als Programm verstehen, welches der Parser des empfangenden Programms ausführen muss, um die Information aus der Nachricht zu extrahieren [BDL+14]. Wenn Nachrichten beliebige Berechnungen im empfangenden Programm auslösen können, ist dies eine Vulnerability, die ein Angreifer ausnutzen kann, um die Kontrolle über das Programm zu übernehmen. Werden maximal kontextfreie Sprachen zur Kommunikation verwendet, die mit kontextfreien Grammatiken definiert werden, lassen sich daraus Parser automatisch generieren, die deterministisch entscheiden ob eine Nachricht zu einer Sprache gehört, bevor Berechnungen auf Basis der Nachricht durchgeführt werden [Aho03].

Auch beim Unparsen von Daten in eine Nachricht muss auf die korrekte Verarbeitung geachtet werden, um Injection Vulnerabilities zu verhindern, da Sprachen eine gewisse Struktur oder Steuerzeichen verwenden, die das korrekte Parsen von Nachrichten ermöglicht. Werden Eingaben eines Programms beim Unparsen direkt in eine Nachrichten übernommen, so können die Eingabedaten diese Nachricht so ändern, dass die Daten beim Parsen durch den Empfänger nicht als Daten, sondern als andere Teile der Nachricht interpretiert werden.

Um diese Vulnerability zu verhindern müssen beim Unparsen alle Eingabedaten eines Programms, die von einem Angreifer kontrolliert werden können entsprechend des Kontextes in dem sie in der zu sendenden Nachricht verwendet werden encoded werden. Durch dieses Encoding werden alle Eingabedaten, die als Teil der Nachricht interpretiert würden in andere, in der Nachricht erlaubte Zeichen kodiert, sodass beim Parsen diese Kodierung erkannt und wieder rückgängig gemacht werden kann. Daten entsprechend des Kontexts in der Nachricht richtig zu kodieren ist in einer Programmiersprache eine schwierige, fehleranfällige Aufgabe, sodass viele Injection Vulnerabilities durch falsches oder fehlendes Encoding entstehen [SML11]. Besonders in komplexen Sprachen wie HTML, die verschiedene Encodings in unterschiedlichen Kontexten verwenden tritt dieses Problem auf, sodass auch viele Frameworks die auf das Unparsen von HTML spezialisiert sind daran scheitern [WSA+11]. Injection Vulnerabilities sind jedoch nicht

auf HTML beschränkt, sondern können in jeder Sprache auftreten, sodass eine Lösung notwendig ist, die für beliebige kontextfreie Sprachen automatisiert anwendbar ist.

Genauso wie Parser lassen sich auch Unparser aus der Definition einer Sprache generieren [Hug95, vdBV96, Wad98]. Diese berücksichtigen jedoch nicht das Encoding, sondern werden als Pretty-Printer bezeichnet, die Dokumente einer Programmiersprache in einer für Menschen besonders lesbaren Form ausgeben.

Neben Security ist Privacy ein wichtiges Qualitätsmerkmal, das Nutzer besonders von einem Cyber Physical System erwarten dem sie ihre Daten zur Verarbeitung übergeben [ZGMW14, ISKC11, TJA10]. Ähnlich wie bei Security existieren grundlegende Prinzipien die allgemein beschreiben wie Privacy bereits im Entwurf eines Systems berücksichtigt werden soll [Cav16, Pea09]. Diese sind jedoch zu vage, sodass Entwickler diese nur schwer umsetzen können [GTD11, MdAY14]. Es gibt eine Vielzahl von Ansätzen die Regeln formulieren, nach denen Nutzerdaten an ein System übertragen werden [BKKF05, BKK06, OAS13, TSR11] oder von ihm verarbeitet werden [KS02, He03, NBL$^+$10, CF12] sollen. Diese sind jedoch nicht in die Softwareentwicklung integriert, sodass Privacy im Entwurf und in der Entwicklung häufig vernachlässigt wird.

All diese Bereiche verbindet, dass zur Lösung der Herausforderungen Entwickler Werkzeuge benötigen, um Dokumente verschiedener Sprachen zu analysieren, transformieren und auszugeben. Um die Erstellung dieser Werkzeuge zu erleichtern existieren verschiedene Language Workbenches [EvdSV$^+$13], von denen hier MontiCore [KRV10, Kra10] verwendet wird, welches speziell für den Einsatz in der agilen Entwicklung ausgelegt ist.

1.3. Forschungsfrage und Herausforderungen

Da Menschen Fehler machen, die durch Angreifer ausgenutzt werden können, ist das Ziel dieser Arbeit Sprachen, Methoden und Werkzeuge zu entwickeln, mit denen Vulnerabilities verhindert oder zumindest (teil-)automatisiert erkannt und behoben werden. Dabei müssen insbesondere Fehler, die im Entwurf in einem agilen modellgetriebenen Entwicklungsprozess entstehen, behandelt und die Privacy der Nutzer bereits in dieser Phase berücksichtigt werden. Somit wird der Frage nachgegangen:

Kann durch Abstraktion in und mit Modellen Software Security und Privacy in IT Systemen leichter eingebaut, nachgewiesen und getestet werden?

Angewandt auf die verschiedenen Aktivitäten eines agilen Softwareentwicklungsprozesses werden die folgenden Problemfelder betrachtet:

- **Security-Architekturbeschreibungssprache**: Ingenieure verwenden zur Konstruktion von technischen Anlagen physikalische Grundeinheiten und bekannte Zusammenhänge zwischen diesen. Bezogen auf die Software-Security-Architektur sind diese Grundeinheiten noch nicht formalisiert und festgelegt und zum Teil vielleicht auch noch nicht bekannt, was

sich in der Entdeckung neuer Angriffsmethoden zeigt. Um Zusammenhänge untersuchen zu können ist es notwendig diese Grundeinheiten möglichst präzise auf dem richtigen Abstraktionsniveau zu definieren und in einer Architekturbeschreibungssprache nutzbar zu machen. Es stellt sich also die Frage: Welche sind diese Grundeinheiten und wie können diese beschrieben werden, sodass Entwickler die Security-Architekturbeschreibung konstruktiv in einem agilen Entwicklungsprozess nutzen können?

- **Security-Architekturanalyse**: Um Vulnerabilities bereits in der Architektur zu erkennen und zu vermeiden müssen diese automatisiert identifiziert werden. Dabei stellt sich die Frage: Wie können bekannte und neue Angriffe formalisiert werden, sodass automatisiert geprüft werden kann, ob ein System gegen diese Angriffe anfällig ist, und welche Maßnahmen zum Schutz gegen diese notwendig sind?
- **Session Management Testen**: Das Session Management von Web Applikationen weist oft bekannte Vulnerabilities auf, die mit Security-Testwerkzeugen nicht automatisiert erkannt werden. Lassen sich diese Vulnerabilities mit einem modellgetriebenen Security Testing Ansatz identifizieren?
- **Verhinderung von Injection Vulnerabilities**: Die Mehrheit der aktuelle entstehenden Vulnerabilities sind Injection Vulnerabilities, welche in immer mehr Systemen und Sprachen identifiziert werden. Daher stellt sich die Frage, wie sich Injection Vulnerabilities für alle kontextfreien Sprachen verhindern lassen?
- **Model-Driven Privacy**: Privacy by Design verfolgt den Ansatz Privacy bereits im Entwurf eines Systems zu etablieren. Im Kontext des Softwareengineering ergeben sich daraus die Fragen: Wie können Entwickler dabei unterstützt werden Privacy in der Entwicklung zu berücksichtigen und wie können Modelle dabei helfen?

1.4. Ergebnisse dieser Arbeit

In dieser Arbeit werden die folgenden Ergebnisse vorgestellt:

- Die Security-Architekturbeschreibungssprache MontiSecArc, welche für den Einsatz durch Entwickler in einem agilen modellgetriebenen Entwicklungsprozess zur konstruktiven Beschreibung der Security-Architektur konzipiert ist.
- Flaw Correction Pattern als eine Notation, die bekannte Vulnerabilities in der Architektur und ihre Lösungen präzise beschreiben, sodass diese automatisiert ohne einen Security-Experten angewandt werden können. In [HHRW15] wurde ein Teil dieser Ergebnisse publiziert.
- Flaw Correction Pattern für bekannte Vulnerabilities und zur Einhaltung von Security-Designprinzipien.
- Generatoren, die die Sprachelemente von MontiSecArc in eine Java Implementierung der spezifizierten Schutzmaßnahmen übersetzen und ein Intrusion Detection System aus der Architekturbeschreibung erzeugen.

- Ein modellgetriebener Security Testing Ansatz, der auf Basis des in einem Statchart modellierten Sollverhalten einer Webapplikation das Session Management dieser Applikation auf bekannte Vulnerabilities testet.
- Ein Ansatz, der Injection Vulnerabilities für alle kontextfreien Sprachen verhindert, indem Encoding als Teil der Sprache in der Grammatik definiert wird und daraus sowohl Parser als auch Unparser generiert werden, die automatisiert kontextspezifisches Encoding und Decoding durchführen. Dieser Ansatz wurde in dem Werkzeug MontiCoder realisiert und in [HKR15] publiziert.
- Ein Model-Driven Privacy Ansatz durch den Entwickler während des Entwurf des Datenmodells die Privacy der Nutzer mit berücksichtigen. Nutzer erhalten in diesem Ansatz die Transparenz und Entscheidung über die Verwendung ihrer Daten durch eine interaktive Privacy Policy, die aus dem Datenmodell generiert wird. In [HHK+14] wurde dieser Ansatz erstmals publiziert und ausführlich im Journalartikel [HHK+16] beschrieben.

1.5. Aufbau der Arbeit

Zunächst werden in Kapitel 2 die grundlegenden Begriffe die in dieser Arbeit verwendet werden definiert und erläutert, sodass im Anschluss die Anforderungen an die Sprachen, Methoden und Werkzeuge in Kapitel 3 definiert werden können.

Im ersten Hauptteil dieser Arbeit werden Modelle zur Abstraktion und Präzisierung von Security in der Entwicklung verwendet. Dabei wird die Security-Architekturbeschreibungssprache MontiSecArc in Kapitel 4 vorgestellt, die durch ihre Sprachelemente die Grundeinheiten für die Beschreibung einer Software-Security-Architektur bereitstellt. Basierend auf MontiSecArc wird in Kapitel 5 beschrieben wie bekannte Flaws in der Security-Architektur zusammen mit einer Lösung als Flaw Correction Pattern dokumentiert werden können, sodass MontiSecArc Modelle automatisiert auf diese Flaws überprüft werden können. In Kapitel 6 wird die Übersetzung von MontiSecArc Modellen in eine Java Implementierung und ein Intrusion Detection System mit Hilfe von Generatoren beschrieben. Der Model-driven Security Test Ansatz mit dem bekannte Vulnerabilities im Session Management von Webapplikationen getestet werden wird in Kapitel 7 beschrieben.

Im zweiten Teil dieser Arbeit wird in Kapitel 8 beschrieben, wie eine Sprache zusammen mit ihrem Encoding definiert wird und aus dieser Definition mit dem Werkzeug MontiCoder ein Unparser generiert wird, der Injections verhindert.

Der Model-Driven Privacy Ansatz wird im dritten Teil in Kapitel 9 beschrieben.

Die Ansätze aus diesen drei Teilen werden in Kapitel 10 evaluiert und ihre Anwendung anhand von Fallstudien aus Projekten demonstriert, bevor Kapitel 11 die Arbeit zusammenfasst und abschließt.

Kapitel 2.

Entwicklungsparadigmen für Security und Privacy

In diesem Kapitel wird die in dieser Arbeit verwendete Security-Terminologie eingeführt, sowie grundlegende Designprinzipien und Prozesse vorgestellt, die zum Entwurf sicherer Systeme verwendet werden. Außerdem wird die agile modellgetriebene Softwareentwicklungsmethodik vorgestellt und die Rolle von Werkzeugen und Sprachen bei dieser Vorgehensweise dargestellt.

2.1. Grundlegende Begriffe

Im Rahmen dieser Arbeit werden englischsprachige Fachtermini anstelle ihrer deutschen Übersetzungen verwendet. Dies dient zum einen der besseren Verbindung mit der verwendeten Literatur und zum anderen, besonders im Bereich der Sicherheits-Terminologie zur präziseren Beschreibung, da im Deutschen der Begriff Sicherheit die Begriffe Safety (Funktionssicherheit) und Security (Informationssicherheit) [Eck14, S. 6] umfasst, hier aber stets Security gemeint ist.

Der Unterschied zwischen Safety und Security liegt darin, dass bei Safety immer von normalen Betriebsbedingungen ausgegangen wird, bei denen alle Systeme und Akteure sich entsprechend der Spezifikation verhalten. Dahingegen werden bei Security auch solche Situationen mit betrachtet, bei denen sich Akteure im System nicht an die vom Entwickler beabsichtigte Spezifikation halten [Eck14, S. 6]. Diese Akteure, auch Angreifer genannt, verletzen die Spezifikation, mit dem Ziel Informationen zu erhalten, zu verändern oder anderen Nutzern den Zugang zum System zu verweigern.

Die wesentlichen (nicht-funktionalen) Security-Eigenschaften eines Systems, welche es zu einem sicheren System machen werden durch die Information Technology Security Evaluation Criteria (ITSEC) [Com91] wie folgt definiert:

Definition 2.1.1 (Security) *Security umfasst die folgenden Eigenschaften:*

Confidentiality - Verhinderung von nicht autorisierter Offenlegung von Informationen;

Integrity - Verhinderung von nicht autorisierter Änderung von Informationen;

Availability - Verhinderung von nicht autorisierter Vorenthaltung von Informationen oder Ressourcen.

Um Availability und damit Security zu erreichen müssen Fragestellungen aus dem Bereich Safety gelöst werden. Allerdings werden, wie eingangs erläutert, in dieser Arbeit die Security-spezifischen Aspekte Confidentiality und Integrity betrachtet.

Definition 2.1.1 legt nahe, dass Schutzmaßnahmen und Funktionen zur Authorization (Autorisierung) notwendig sind, um Security in einem System zu realisieren. So werden Zugriffsrechte definiert, die festlegen welche Nutzer, auch Subjekte genannt, welche Art von Zugriff auf Objekte haben. Dabei ist ein Objekt eine von einem System zu schützende Information oder Funktion.

Definition 2.1.2 (Authorization) *Authorization bezeichnet den Prozess in einem System, der auswertet ob die Zugriffsrechte eines Subjekts auf ein Objekt ausreichen und somit der Zugriff gewährt wird oder nicht. [Eck14, S. 5]*

Eine naheliegende Möglichkeit eine Authorization zu umgehen, ist dass das Subjekt sich als ein anderer Nutzer ausgibt und somit dessen Zugriffsrechte erlangt. Daher wird vor der Authorization eine Authentication des Subjekts durchgeführt.

Definition 2.1.3 (Authentication) *Bei der Authentication wird nachgewiesen, dass ein Subjekt tatsächlich ist, wer es behauptet. [Eck14, S. 8]*

In verteilten Systemen wird die Authentication mittels kryptographischer Protokolle durchgeführt [NS78] und die Authorization durch die Programmlogik umgesetzt. Für die Realisierung der Authorization gibt es verschiedene Schemata, die zur Definition und Überprüfung von Nutzerrechten verwendet werden. Role-Based Access Control (RBAC) [FKC07, SS94] und Access Control Lists (ACL) [SS94] gehören zu den am meisten verwendeten Schemata. Bei RBAC werden alle Rechte in einem System, die ein Subjekt zur Erfüllung einer bestimmten Aufgabe benötigt zu einer Rolle zusammenfasst. Ein Nutzer, der eine oder mehrere Aufgaben mit dem System erfüllen soll, erhält genau die Rollen zugewiesen, die seinen Aufgaben entsprechen. Um zu entscheiden, ob ein Subjekt Zugriff auf ein bestimmtes Objekt hat prüft das System, ob eine der Rollen, die das Subjekt besitzt Zugriffsrechte hat.

Im ACL Schema hingegen werden alle Rechte, die ein Subjekt an einem Objekt hat in einer Access Control Liste gespeichert, die typischerweise zusammen mit dem Objekt, also beispielsweise einer Datei gespeichert wird. Fragt ein Subjekt den Zugriff auf ein Objekt an, so entnimmt das System dieser Liste die Rechte, welche das Subjekt auf das Objekt hat und entscheidet dementsprechend über den Zugriff.

ACLs erlauben also eine sehr individuelle Definition von Rechten für einzelne Objekte im Vergleich zu RBAC. In RBAC lassen sich allerdings durch die Vergabe von Rollen an Subjekte Berechtigungen besser zentral vergeben und auch wieder entziehen als mit ACLs, da bei ACLs nur durch eine Suche über alle ACLs zu ermitteln ist, auf welche Objekte ein Subjekt Zugriff hat. Um die Vorteile beider Schemata zu kombinieren, werden diese daher häufig zusammen verwendet, sodass nicht die Rechte von Subjekten, sondern die von Rollen in den ACLs gespeichert werden.

Ein Ansatz zu definieren, ob ein System *sicher* ist, besteht darin Regeln in einer Policy aufzustellen und zu überprüfen, ob diese im gesamten System gelten. Eine solche Policy stellt das Bell-LaPadula Modell [BL73] dar, welches jedem Subjekt und Objekt ein Security Level zuweist. Security Level sind dabei, wie bei der Geheimhaltung von Informationen üblich, z.B. öffentlich, geheim und streng geheim. Nach dieser Multi Level Security (MLS) Policy erfüllt ein System die Confidentiality, wenn es folgende Eigenschaften jederzeit sicherstellt:

- Ein Subjekt darf Dokumente lesen, wenn das Security Level des Objekts kleiner oder gleich dem des Subjekts ist.

- Ein Subjekt darf Dokumente schreiben, wenn das Security Level des Objekts größer oder gleich dem des Subjekts ist.

2.1.1. Software Security

Im Bezug auf Software gibt es im Security-Bereich häufig eine Verwechselung zwischen Security Software, Application Security und Software Security. Security Software bezeichnet Programme, die Maßnahmen wie Authorization, Authentication oder Erkennung von bekannten Computerviren realisieren. Application Security ist ein Prozess mit dem Ziel Software abzusichern, der nach der Erstellung der Software beginnt und üblicherweise durch Konfiguration und hinzufügen von Security Software durchgeführt wird. Im Jahr 2014 wurde mit dem Verkauf von Security Software ein Umsatz von 21 Milliarden US-Doller erzielt [Gar15], daher könnte man schlussfolgern, dass Security ein gut verstandenes Forschungsgebiet ist, welches bereits erfolgreiche Produkte hervorgebracht hat und als gelöst betrachtet werden kann. Allerdings reicht es nicht aus alle verfügbaren Security Software Produkte einzusetzen, um ein entwickeltes System abzusichern, da diese Produkte nicht alle Aspekte von Security in Software abdecken, nicht kompositionsfähig sind und sich unter Umständen sogar gegenseitig bei der Erreichung des Ziels behindern [LBS09].

In Abgrenzung dazu definiert McGraw in seinem gleichnamigen Buch Software Security als:

Definition 2.1.4 (Software Security) *Der Prozess Software mit dem Fokus auf Security zu entwerfen, bauen und testen, sodass Software entsteht die Angriffen widersteht [McG06, S. 14].*

Bei Software Security wird im Gegensatz zu Application Security bereits während der Softwareentwicklung Security mit betrachtet und eingebaut, sodass die im Entwicklungsprozess erhobene Anforderungen, getroffen Entscheidungen und erstellte Implementierungen den Security-Aspekt berücksichtigen.

Da sich Angreifer jedoch nicht an die Spezifikation von Systemen und halten, sondern gerade die Unterschiede zwischen intendierter Spezifikation und Verhalten des fertigen Systems ausnutzen, um die Security zu kompromittieren profitieren sie von einer Vielzahl von Fehlern, die im Laufe eines Softwareentwicklungsprozesses entstehen. Es ist die Aufgabe von Software Security diese Fehler zu erkennen und zu beheben, oder besser noch zu verhindern. Dabei gilt es verschiedene Arten von Fehlern zu unterscheiden:

Definition 2.1.5 (Weakness) *Eine Weakness (Schwachstelle) ist eine Schwäche eines Systems, an der das System verwundbar werden kann. [Eck14, S. 16]*

Definition 2.1.6 (Vulnerability) *Eine Vulnerability (Verwundbarkeit) ist eine Weakness, die ausgenutzt werden kann um Security-Eigenschaften eines Systems zu umgehen. [Eck14, S. 16]*

Eine Weakness ist demnach eine Abweichung des tatsächlichen Systemverhaltens von der intendierten Spezifikation, die von einem Angreifer vielleicht ausgenutzt werden könnte, um unter Umständen eine Security-Eigenschaft des Systems zu verletzen. Wenn eine solche Weakness tatsächlich dazu verwendet werden kann, um eine Security-Eigenschaften des Systems zu kompromittieren, wird sie als Vulnerability bezeichnet.

Für die weitere Diskussion ist ebenfalls von Interesse auf welcher Abstraktionsstufe der Systemspezifikation die Weakness auftritt:

Definition 2.1.7 (Bug) *Ein Bug ist eine Weakness, die durch einen Fehler in der Implementierung verursacht wird. [McG06, S. 14]*

Definition 2.1.8 (Flaw) *Ein Flaw ist eine Weakness, die durch einen Fehler in einem Design Dokument verursacht wird. [McG06, S. 16]*

Wie Boehm und Basili [BB01] festgestellt haben, steigen die Kosten zur Fehlerbehebung, je früher im Entwicklungsprozess ein Fehler passiert. Dies gilt insbesondere auch für Flaws, denn um einen Flaw in einem fertig implementierten System zu beheben müssen verschiedene Teile der Implementierung angepasst werden, die von der Entwurfsentscheidung betroffen sind damit an keiner Stelle im Code eine Vulnerability offen bleibt.

An der Entwicklung von sicheren Systemen sind neben Entwicklern in der Regel auch Auditoren beteiligt. Ein Auditor betrachtet das System aus dem Blickwinkel eines Angreifers mit dem Ziel Weaknesses zu identifizieren und diese zusammen mit möglichen Schutzmaßnahmen an die Entwickler zur Umsetzung zu geben. Entwicklern sind daran interessiert ein System zu konstruieren und die funktionalen Anforderungen umzusetzen. Daher liegt es Entwicklern nicht wie ein Auditor oder Angreifer zu denken und das mühsam erstellte System zu hinterfragen, kritisieren und Vulnerabilities zu identifizieren [McG06]. Auditoren sind typischerweise keine Entwickler und auch nicht an der Konstruktion eines Systems beteiligt, um ein System möglichst unvoreingenommenen auditieren zu können.

Um eine Brücke zwischen Unternehmerischen Größen wie möglichem monetärem Verlust und technisch bestimmten Größen wie Vulnerabilities und Schutzmaßnahmen eine Verbindung herzustellen wird Risk Management eingesetzt [Gee03]. Dieses ist ein Management Werkzeug, mit dem begründet werden kann warum Unternehmen Geld für Security ausgeben sollten.

Der Zentrale Begriff Risk ist dabei wie folgt definiert:

Definition 2.1.9 (Risk) *Risk = $E \times A$*
mit E = Eintrittswahrscheinlichkeit eines Schadensereignisses
und A = Auswirkungen auf ein Gut [McG06, S. 12], [Nat13]

Angewendet auf Finanziellen Verlust ergibt sich:

Definition 2.1.10 *Jährlicher Verlust = Kosten eines Vorfalls $\times$ Vorfälle pro Jahr [McG06, S.152]*

Somit hilft Risk Management auf Management Ebene zu entscheiden, ob und wie viel in die Abwehr von Angriffen beispielsweise durch Software Security investiert werden soll.

Da es schwierig ist verlässliche Metriken zu finden, die die Existenz von Weaknesses in einem System bestimmen [PC10], ist somit auch die Vorhersage der Eintrittswahrscheinlichkeit schwierig. Weaknesses werden in dieser Arbeit daher nicht mit einem probabilistischen Risk Ansatz betrachtet, sondern entsprechend ihrer Ursachen in der Entwicklung untersucht.

2.1.2. Privacy

Ein weiterer Begriff, der oftmals mit Security zusammen als nichtfunktionale Eigenschaft eines Systems genannt wird ist Privacy, die auch beschrieben wird, als “the right to be forgotten” [Ros12].

Definition 2.1.11 (Privacy) *Privacy (Datenschutz) bezeichnet die Fähigkeit einer Person, die Weitergabe von Informationen, die sie persönlich betreffen, zu kontrollieren. [Eck14]*

Das europäische bzw. deutsche Datenschutzgesetz sichert jedem Bürger eben diese Privacy zu, woraus folgt das Systeme die personenbezogene Daten verarbeiten konform zu diesen Gesetzen sein müssen und die Nutzeranforderungen an Privacy erfüllen müssen. Diese Anforderungen der Nutzer an ein System sind nach Pearson [Pea09]:

1. **Ankündigung (notice):** Bevor ein System Daten über einen Nutzer erhebt muss der Nutzer darüber informiert werden, dass dies geschieht. Außerdem muss dem Nutzer verständlich dargelegt werden welche Daten, zu welchem Zweck erhoben werden und an wen sie weitergegeben werden.

2. **Zustimmung (consent):** Nutzer müssen die Wahl haben der Sammlung ihrer Daten zuzustimmen oder dies abzulehnen und das System muss diese Entscheidung umsetzen.

3. **Selbstbestimmung (self-determination):** Nutzer müssen in der Lage sein den Umgang mit ihren Daten zu überprüfen und auch die Richtigkeit und den Umfang der erhobenen Daten kontrollieren können.

4. **angemessene Security (adequate security):** Die Organisation, welche das System betreibt ist verantwortlich die personenbezogenen Daten adäquat zu schützen. Dieser Schutz beginnt auf organisatorischer Ebene, indem ein Datenschutzverantwortlicher bestimmt wird und reicht bis zu technischen Security-Maßnahmen, die den Zugriff auf die Daten verhindern.

5. **zweckgebunden Nutzung (purposeful use):** Die Nutzung der Daten muss auf den einmal vom Nutzer festgelegten Zweck beschrenkt bleiben. Dies beinhaltet auch, dass personenbezogene Daten nur veröffentlicht oder an Dritte weitergegeben werden dürfen, wenn der Nutzer dem zugestimmt hat.

Um die Privacy von Nutzern zu erreichen ist es also notwendig die Security eines Systems sicherzustellen, aber noch darüber hinaus organisatorische und technische Maßnahmen zu etablieren, um die personenbezogenen Daten zu schützen. Privacy in ein System einzubauen, dass bereits in Betrieb ist, ist noch schwieriger als Security nachträglich einzubauen, denn einem Servicenutzer glaubhaft zu versichern, dass einmal von einem Service erhobene Daten wieder gelöscht werden, obwohl der Serviceprovider offenkundig ein Interesse an diesen Daten hat ist sehr schwierig. Daher setzt Privacy-by-Design [Cav16] analog zu Software Security darauf, dass bereits während der Systementwicklung Privacy im Entwurf des Systems berücksichtigt wird.

2.1.3. Reaktives Vulnerability Management

Vulnerabilities werden oftmals erst nach der Auslieferung eines Systems durch Nutzer oder Security-Experten gefunden. Somit sind Systeme, die diese Vulnerability besitzen bereits im Einsatz und deren Betreiber bzw. Nutzer sollten über die Vulnerability unterrichtet werden, damit sie Maßnahmen ergreifen können um eine Ausnutzung der Vulnerability zu verhindern. Hierbei kann

der Nutzer auch ein anderes System sein, welches aufgrund dieser Vulnerability unter Umständen angepasst werden muss. Zwar ist eine Aktualisierung der Software, die die Vulnerability direkt behebt der bevorzugte Lösungsweg, allerdings dauert es auch bei namhaften Softwareproduzenten wie die SAP AG mitunter eineinhalb Jahre, bis diese zur Verfügung steht [Sil11] [1]. Wenn die Ursache der Vulnerability nicht zeitnah behoben werden kann, bleibt nur die Möglichkeit gezielt den betroffenen Teil der Software abzuschalten, wenn er nicht benötigt wird. Alternativ kann einen Filter vor der Vulnerability etabliert werden, der Angreifer davon abhält die Vulnerability auszunutzen oder die Auswirkungen einer erfolgreichen Ausnutzung der Vulnerability können minimiert werden. All dies sind reaktive Maßnahmen, die je nach Vulnerability individuell angepasst werden müssen. Bekannte reaktive Maßnahmen aus dem Bereich Security Software sind Virenscanner, Intrusion Detection Systeme und Web Application Firewalls. Um Vulnerabilities managen zu können und nachzuvollziehen, welche Maßnahme getroffen wurden um die Ausnutzung von Vulnerabilities zu verhindern und Klarheit zu schaffen, welche Vulnerabilities eine Softwareaktualisierung behebt, wird jede Vulnerability durch eine Common Vulnerabilities and Exposures ID (CVE) eindeutig identifiziert, die von der MITRE vergeben wird. Zu jeder CVE werden durch die National Vulnerability Database (NVD) [Nat15] weitere Metadaten gespeichert wie die Common Platform Enumeration (CPE), die maschinenlesbar beschreibt welche Produkte die Vulnerability beinhalten, und den mit dem Common Vulnerability Scoring System (CVSS) ermittelten Wert für die Schwere der Vulnerability. Diese Informationen werden genutzt, um ganze IT Landschaften anhand der Informationen über die eingesetzten Produkte nach bekannten Vulnerabilities abzusuchen, was angesichts der steigenden Zahl von CVEs nur automatisiert möglich ist.

2.1.4. Entwicklung sicherer Software

Dem Prinzip folgend aus bekannten Fehlern zu lernen, sind eine ganze Reihe von Sammlungen, Checklisten und Büchern entstanden mit denen Entwickler darin geschult werden, wie sie bestehende Technologien richtig verwenden, um sichere Systeme zu entwickeln.

Secure Coding [Sea13, LMS+13, LMS+11] zielt dabei darauf ab Entwicklern die unintuitiven und leicht übersehbaren Details von Programmiersprachen zu vermitteln, durch die Vulnerabilities bei der Implementierung entstehen.

Die Common Weakness Enumeration (CWE) [MIT15b] und Common Attack Pattern Enumeration and Classification (CAPEC) [MIT15a], welche genau wie die CVE vom MITTRE organisiert werden, zielt darauf ab jeder Arten von Weakness eine CWE ID und jeder Art von Angriff eine CAPEC-ID zu geben. Diese werden hierarchisch strukturiert, um so das Wissen über Weaknesses und Angriffe zu organisieren. Zu jeder Weakness wird neben einer Beschreibung ein Beispiel angegeben, welches die Weakness anhand von Quellcode demonstriert.

Das IEEE Center for Secure Design (CSD), welches im Jahr 2014 gegründet wurde verfolgt nun einen ähnlichen Ansatz, allerdings speziell für die Systemarchitektur. Im ersten Schritt wurden bekannte Flaws gesammelt und zu einer Top 10 Liste zusammengestellt wurden [IAD+14].

[1] Im konkreten Fall betraf die Vulnerability den NetWeaver Applicationserver, auf dem alle Webanwendungen aus dem Portfolio an Business Applikationen der SAP AG aufbauen. Da SAP 9 Jahre lang Aktualisierungen für Produkte bietet, musste die Vulnerability in mehreren Releases behoben und in verschiedensten Konfigurationen getestet werden bevor eine Aktualisierung veröffentlicht werden konnte.

In Zukunft möchte das CSD Hilfen entwickeln, durch die bekannte Flaws bei Architekturentscheidungen vermieden werden sollen.

Dem Pattern Ansatz der "Gang of Four" folgend existieren allerdings bereits viele Security Pattern [SFBH+13], die für viele Security-Probleme eine Lösung angeben. Diese Pattern beschreiben dabei zum großen Teil wie Security-Funktionen in objektorientierten Programmiersprachen implementiert werden können und sind damit wesentlich näher an der Implementierung als die vom Center for Secure Design beschriebenen Flaws.

Ein leichtgewichtigerer Ansatz, speziell zur Überprüfung der Sicherheit von Web-Applikationen wurde vom Open Web Application Security Project (OWASP) entwickelt, welches ins Leben gerufen wurde, um Entwicklern von Webapplikationen Security-Knowhow zu vermitteln. Dieser OWASP Testing Guide [Ope15c] liegt mittlerweile in Version 4 vor und besteht im Kern aus einer Checkliste, die alle bekannten Angriffstechniken auf Webapplikationen aufzählt und kategorisiert. Diese umfangreiche Liste nutzen Penetrationstester, um ihrem kreativem, explorativem und sehr individuellen Vorgehen ein gewisses Maß an Reproduzierbarkeit zu verleihen, indem sie dieser Checkliste folgen und alle dort aufgeführten Angriffstechniken durchführen. Der Testing Guide ist als Anleitung zum Einstieg in die Überprüfung der Sicherheit von Webapplikationen konzipiert und enthält neben einer Beschreibung der Angriffstechniken auch Hinweise auf Werkzeuge, die zur teil-automatisierten Prüfung verwendet werden können. In Version 4 wird erstmals eine methodische Einleitung in die Vorgehens- und Denkweise von Penetrationstestern gegeben und die Integration in einen Entwicklungsprozess beschrieben.

Als Anleitung zur Entwicklung von sicheren Webapplikationen und als Gegenstück zum OWASP Testing Guide gibt es den OWASP Developer Guide [Ope15b], der beschreibt wie die sicherheitsrelevanten Schutzmechanismen einer Webapplikation implementiert werden sollten. Das knapp 300 Seite starke Dokument gibt einen sehr breiten Überblick von Security-Managementprozessen über Techniken zur Abwehr von Angriffen wie Phishing bis hin zu Gegenmaßnamen gegen verschiedene Injection Angriffe.

Anstatt diese technologiespezifischen Hinweise zu verwenden, lassen sich die Security-Eigenschaften Confidentiality und Integrity auch als allgemeingültige Anforderungen an ein System verstehen. Da diese jedoch sehr allgemein sind und orthogonal zu allen funktionalen Anforderungen liegen und damit in jeder Entwurfsentscheidung berücksichtigt werden müssen ist dies ohne weitere Hilfsmittel nicht praktikabel. Daher entwickelten Saltzer und Schroeder Prinzipien für den Entwurf sicherer Systeme [SS75], die bereits bei der Erstellung der Architektur angewandt werden und Viega und McGraw aktualisierten und erweiterten diese [VM02, S. 91ff]. Die wesentlichen Security-Designprinzipien, die auch in dieser Arbeit Anwendung finden sind:

1. **Secure the Weakest Link** [VM02, S. 93]: Security wird oft als Kette bezeichnet, die nur so stark ist wie ihr schwächstes Glied, da Angreifer genau diese schwächste Stelle für einen Angriff ausnutzen. Dieses Designprinzip besagt, dass im Entwurf die schwächste Stelle identifiziert werden soll, die eher selten in anerkannten Standards beispielsweise für Verschlüsselung liegen, sondern viel öfter im Programmcode, der die Daten verarbeitet und noch nie einem Code Review unterzogen wurde. Um die Sicherheit des Gesamtsystems zu verbessern muss dementsprechend diese schwächste Stelle verbessert werden.

2. **Defense in Depth** [VM02, S. 96]: Die grundsätzliche Beobachtung hinter diesem Prinzip ist, dass eine Schutzmaßnahme (genauso wie jede Software) Fehler enthalten kann, sodass sie ihre Schutzfunktion nicht durchführt. Damit eine einziger Fehler nicht dazu führt, dass das System komplett kompromittiert werden kann besagt das Prinzip *Defense in Depth* dass im Falle einer Fehlfunktion einer Schutzmaßnahme weitere Schutzmaßnahmen einen Angreifer daran hindern sollen das System zu kompromittieren.

3. **Fail-safe Defaults** [SS75]: Eine Policy, die beispielsweise den Zugriff auf ein System regelt, sollte auf der Grundlage definiert werden, dass alles verboten ist und darauf aufbauend auflisten unter welchen Bedingungen der Zugriff erlaubt ist. Wird in einer solchen Policy ein Fehler gemacht, fällt dieser schneller auf, da der Nutzer für den eine Änderung an der Policy vorgenommen wurde das System nicht wie erwartet nutzen kann. Wohingegen ein Fehler in einer Policy die nur Zugriffsverbote auflistet nicht auffällt, da im normalen Betrieb niemand überhaupt Versucht auf auf Ressourcen zuzugreifen auf die er ohnehin keinen Zugriff haben sollte.

4. **Least Privilege** [SS75] [VM02, S. 100]: Jeder Komponente eines Systems sollten nur die Zugriffe auf Ressourcen gewährt werden, die sie zur Erfüllung ihrer Aufgaben benötigt. Wenn eine Komponente mehr als die notwendigen Ressourcen besitzt ist das Gesamtsystem einem höheren Risiko ausgesetzt, da im Falle einer Kompromittierung dieser Komponente ein Angreifer die Kontrolle über mehr Ressourcen erlangt. Daher zielt *Least Privilege* darauf ab die Auswirkungen eines Angriffs zu minimieren.

5. **Compartmentalization** [SS75] [VM02, S. 102]: *Least Privilege* setzt voraus, dass es unterschiedliche Rechte in einem System gibt, die gewährt werden können. Unterschiedliche Rechte sind nur sinnvoll, wenn diese auch durchgesetzt werden können, sodass ein Akteur mit einem Recht nicht direkt auch andere Rechte erlangt. Durch *Compartmentalization* wird ein System derart in Einheiten unterteilt, sodass für diese einzelne Rechte vergeben werden können. Saltzer bezeichnet dies als *Least common mechanism* und betont vor allem, dass die einzelnen Einheiten möglichst schmale, wohldefinierte Schnittstellen haben müssen.

6. **Be Reluctant to Trust** [VM02, S. 111]: Vertrauensbeziehungen in einem System sollten kritisch hinterfragt werden, da diese oftmals von Angreifern ausgenutzt werden, um mehr Zugriff zu erlangen. Beispielsweise sollten einzelne Teile eines Systems, die durch *Compartmentalization* getrennt werden sich nicht gegenseitig vertrauen, damit im Falle einer Kompromittierung eines Teils ein Angreifer keine Kontrolle über einen anderen Teil erlangt. Dieses Prinzip gilt aber nicht nur für technische Systeme, sondern auch für Menschen. Sodass Social Engineering Angriffe, die das Ziel haben Zugang zu Gebäuden und Informationen zu bekommen, ausnutzen, dass Menschen hilfsbereit gegenüber Fremden sind und ihnen vertrauen.

7. **Separation of Privilege** [SS75]: Um die Wirkung einer Authorization zu verstärken und die versehentliche oder missbräuchliche Nutzung organisatorisch zu erschweren sieht *Separation of Privilege* vor, dass für eine erfolgreiche Authorization zwei Personen sich

authentisieren müssen. Sind diese Personen unabhängig voneinander, haben also keinerlei Weisungsbefugnis über den Anderen wird die Aktion nur dann durchgeführt, wenn beide Personen diese auch für richtig halten. *Separation of Privilege* wird zur Authorization von besonders kritischen Aktionen, die nicht rückgängig gemacht werden können eingesetzt wie z.B. beim Start von Atomaren Raketen, aber auch bei Freigabeprozessen in Businessanwendungen

8. **Keep It Simple** [SS75] [VM02, S. 104]: Eine komplexe Lösungen ist schwieriger zu verstehen, wodurch die Wahrscheinlichkeit, dass Fehler durch Reviews gefunden werden sinkt. Komplexe Lösungen neigen daher dazu subtile Security Vulnerabilities zu enthalten. Das Prinzip *Keep It Simple* fordert daher möglichst einfache, klar verständliche Lösungen zu entwickeln. Dazu gehört unter Anderem, dass Wiederverwendung betrieben wird und Sicherheitsmaßnahmen an einigen wenigen zentralen Stellen im System angewandt werden, sodass einfach ersichtlich ist, dass das gesamte System geschützt ist.

9. **Open Design** [SS75] [VM02, S. 110]: Geheimnisse sind nur sehr schwer wirklich geheim zu halten, daher sollte die Sicherheit eines Systems nicht darauf beruhen, dass ein Angreifer diese nicht herausbekommt. Die Sicherheit eines Systems darf daher nicht davon abhängen, dass der Angreifer die Funktionsweise eines Systems nicht kennt.[2] Genauso muss in einem System vorgesehen sein, dass Geheimnisse wie Passwörter oder Schlüssel zur Verschlüsselung und Signatur kompromittiert werden können. Daher müssen solche Geheimnisse auch austauschbar sein und können nicht einfach fest einmalig im Quellcode gesetzt werden. Abgesehen davon kann ein Angreifer diese Schlüssel aus dem ausgelieferten Programm extrahieren.

10. **Complete Mediation** [SS75]: Sicherheitsmechanismen wie Zugriffskontrolle müssen konsequent im gesamten System angewandt werden, überall dort wo ein Zugriff stattfindet. Diese Sicherheitsmechanismen müssen so narrensicher sein, dass weder Nachlässigkeit noch Unwissenheit bei der Entwicklung dazu führen, dass sie nicht eingesetzt werden.

2.2. Secure Development Lifecycle

Security beschränkt sich jedoch nicht auf die zuvor erläuterten technischen Eigenschaften von Systemen bzw. Software, sondern umfasst den gesamten Lebenszyklus eines Systems, sowie die Verantwortlichkeiten und Prozesse, welche dazu führen, dass ein System entwickelt und betrieben wird.

2.2.1. ISO Security-Managementprozesse

Gemäß ISO 27001 [ISO13] ist Security ein Management Prozess innerhalb eines Unternehmens. Um diesen zu organisieren beschreibt ISO 27001 ein Information Security Management System (ISMS), welches das Ziel hat organisatorische und technische Maßnahmen in einem Unternehmen

[2] Wird dieses Prinzip verletzt spricht man auch von *Security through Obscurity*

zu etablieren, überprüfen und zu verbessern, die Security-Risiken von den Werten des Unternehmen abwehren. Die zugehörigen Maßnahmen, die in ISO 27002 beschrieben werden, richten sich primär an das Management des Betriebs von IT Systemen und nur im Kontext von Beschaffung und Unternehmensübernahmen an die Entwicklung neuer Systeme.

Speziell für die Security von Anwendungen beschreibt ISO 27034 [ISO11a] einen Application Security Management Prozess, der die Erstellung, den Betrieb und die Wartung von Software Applikationen als Ganzes betrachtet. Dieser Prozess bietet dabei die Grundstruktur um Application Security Controls (ASC) zu definieren, sie in den Managementprozess einzubringen und deren Wirksamkeit mittels Verification Measurement zu kontrollieren. Eine Security Activity beschreibt dabei das konkrete Vorgehen, welches in einem ASC durchgeführt wird, um Komponenten nach ihrer Entwicklung mit Security-Maßnahmen zu sichern oder Aktivitäten, die in den Lebenszyklus der Applikation eingebunden werden, um die Security der Applikation zu verbessern.

2.2.2. Microsoft Security Development Lifecycle

Nachdem Kunden von Microsoft Produkten nach mehr Security verlangten wurde der Microsoft Security Development Lifecycle (SDL) [HL06] eingeführt. Im Kontrast zum top-down Ansatz der ISO besteht dieser aus 13 sehr praktischen Phasen, die bei Microsoft angewendet werden und zu Software mit besseren Security-Eigenschaften geführt haben [HL06]. Der SDL beschreibt, wie Security durch Management Commitment und Schulungen in bestehende Teams eingeführt wurde und wie die Phasen des SDL in den bestehenden Entwicklungsprozess eingebunden wurden. Für jede dieser 13 Phasen beschreibt der SDL mehrere Aktivitäten und Hilfsmittel, die zusätzlich zur eigentlichen Entwicklung angewandt werden.

Für die Entwurfsphase werden im SDL neben den bereits genannten Prinzipien für den Entwurf sicherer Systeme von Saltzer und Schroeder [SS75], eine Attack Surface Analyse und Reduktion, Threat Modeling und der STRIDE Ansatz verwendet.

Die Attack Surface Analysis [HL06, S. 78] wird motiviert aus Saltzer and Schroeders Prinzipien Least Privilege und Economy of Mechanism. Dabei ist der Attack Surface die Vereinigung allen Codes, Interfaces, Diensten und Protokollen eines Systems, welche durch nicht authentisierte Nutzer (bzw. Angreifer) erreichbar ist. Da Systeme immer Weaknesses haben, ist das Ziel einer Attack Surface Reduktion einem Angreifer möglichst wenig Angriffsfläche, also Code, der Weaknesses enthält zugänglich zu machen. Somit werden existierende Weaknesses nicht zu Vulnerabilities, da sie nicht ausgenutzt werden können. In Microsoft Produkten werden daher Funktionen, die nicht von allen Nutzern benötigt werden im Auslieferungszustand deaktiviert und aktive Funktionen müssen ein höheres Maß an Qualität aufweisen.

Threat Modeling [HL06, S. 103], [Sho14], welches von McGraw als Architectural Risk Analysis [McG06] bezeichnet wird, da nicht die Bedrohungen (Threats) eines Systems modelliert werden, sondern seine Architektur, verwendet Data Flow Diagrams (DFDs) zur Architekturbeschreibung. In einem DFD werden die Prozesse und Datenspeicher eines Systems, externe Entitäten, Datenflüsse zwischen diesen und die Systemgrenzen zusammen darzustellen.

Abbildung 2.1 zeigt ein DFD in dem der Prozess Printer Service, der Datenspeicher Log, sowie die externen Entitäten Admin, Human User und Printer dargestellt sind. Außerdem werden die gerichteten Datenflüssen zwischen diesen Elementen mit Pfeilen, sowie die Trust Bounderies als

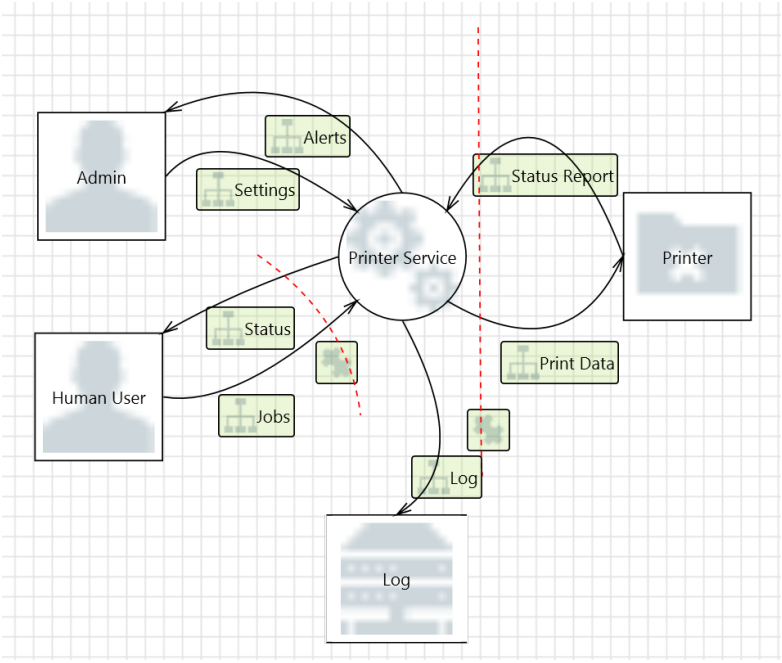


Abbildung 2.1.: Dieses Data Flow Diagram beschreibt die Datenflüsse zwischen einem Printer Service und seinen Nutzern.

rote gestrichelte Linien dargestellt. Eine Trust Boundary ist dabei eine Systemgrenze an der die Kontrollfunktionen eines Systems wie beispielsweise Authorization enden.

Um Weaknesses in einem System zu finden wird nun die STRIDE Methode [Sho14, S. 61] auf alle Elemente des DFD (STRIDE-per-Element) oder alle Datenflüsse (STRIDE-per-Interaction) angewendet. STRIDE ist ein Akronym, welches für die Angriffe Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service und Elevation of Privilege steht. Beim Spoofing versucht der Angreifer die Identität eines legitimen Nutzers anzunehmen und das System mit dessen Rechten zu benutzen. Er nutzt also eine fehlende oder schwache Authentication aus. Tampering bezeichnet die Veränderung von Daten, welche von einem System verarbeitet werden. Dabei versucht der Angreifer das Verhalten des Systems mithilfe der veränderten Daten zu seinem Vorteil zu verändern und so Zugriff zu erhalten. Hier nutzt der Angreifer eine mangelnde syntaktische Prüfung von Eingabedaten aus. Repudiation erlaubt es einem Angreifer Angriffe auf ein System durchzuführen, ohne dass dies durch das System protokolliert wird. Somit kann der Angriff längere Zeit unerkannt bleiben und es ist dem Angreifer möglich Rateangriffe auf Geheimnisse wie Passwörter unerkannt durchzuführen. Bei Information Disclosure ist es einem Angreifer möglich Informationen zu lesen, auf die er keinen Zugriff haben sollte. Dieser Zugriff kann z.B. bei der Übertragung über eine Netzwerkverbindung erfolgen oder durch falsch eingestellte Zugriffskontrolle von Dateien. Der Angreifer nutzt hierbei aus, dass Informationen nicht ausreichend durch ein Berechtigungssystem oder Verschlüsselungsalgorithmen geschützt werden. Ein Denial of Service (DoS) Angriff nutzt die kritischen Ressourcen eines Systems, sodass legitime Nutzer des Systems wegen einem Mangel an Ressourcen abgewiesen werden müssen. Dabei nutzt der

Angriff	Security-Eigenschaft
Spoofing	Authentication
Tampering	Integrity
Repudiation	Non-Repudiation (Integrity)
Information Disclosure	Confidentiality
Denial of Service	Availability
Elevation of Privilege	Authorization

Tabelle 2.2.: Der Zusammenhang zwischen den Angriffen der STRIDE Methode und Security-Eigenschaften bzw. Schutzmaßnahmen

Angreifer aus, dass ein System für die Bearbeitung von Anfragen Ressourcen dauerhaft reserviert bzw. nicht rechtzeitig wieder freigibt und nicht unterscheiden kann. Elevation of Privilege zielt darauf ab, dass ein Angreifer Aktionen mit höheren als die ihm zugeordneten Rechten durchführt. Diese Situation kann z.B. durch einen Fehler in der Authorization hervorgerufen werden.

Jeder dieser Angriffe versucht eine Security-Eigenschaft an einer Trust Bounderies des im DFD modellierten Systems zu verletzen. Der Zusammenhang zwischen Angriff und zu verletzender Eigenschaft ist in Tabelle 2.2 dargestellt. Beispielsweise würde ein Spoof Angriff auf das DFD aus Abbildung 2.1 zwischen Human User und Printer Service eine Weakness identifizieren, wenn im DFD nicht vermerkt ist, dass zwischen diesen beiden eine Authentication stattfindet.

Mit Hilfe des Microsoft Threat Modeling Tool [Mic15] lassen sich DFDs erstellen und STRIDE automatisiert darauf anwenden. Zur Erstellung der DFDs stehen vorgefertigte generische und technologiespezifische Elemente zur Verfügung, die über Eigenschaften verfügen aus denen sich ableitet, ob diese Elemente in der STRIDE Analyse als verwundbar identifiziert werden. Die Verbindung HTTPS hat z.B. die Eigenschaften Destination Authenticated, Confidentiality und Integrity. Die automatische STRIDE Analyse liefert eine Liste von Bedrohungen, die die Auswirkung eines Angriffs und eine mögliche Gegenmaßnahme beschreiben. Außerdem erlaubt es diese Liste zu dokumentieren, wie diese Bedrohungen im weiteren Verlauf der Entwicklung behandelt wurden. So wird jede Statusänderung einer Bedrohung z.B. zu nicht anwendbar oder abgewehrt mit Zeitstempel und Benutzername des Ändernden gespeichert.

2.2.3. Security Touchpoints

McGraw beschreibt, ähnlich wie der SDL von Microsoft sieben Touchpoints [McG06], mit denen ein Softwareentwicklungsprozess sicherere Software produzieren soll. Er priorisiert diese Aktivitäten gemäß ihrer Wirksamkeit und stuft dabei Architectural Risk Analysis auf Platz zwei, direkt nach Code Review mit Hilfe von Statische Code Analyse ein. Architectural Risk Analysis wird auf Basis einer Architekturübersicht durchgeführt, die Komponenten und Module des Systems zeigt. Flaws werden durch drei Aktivitäten identifiziert:

1. In einer *Attack Resistance Analysis* werden wie bei STRIDE bekannte Angriffstechniken auf die Architektur angewandt.

2. Durch eine *Ambiguity Analysis*, bei der mehrere Auditoren unabhängig voneinander versu-

chen das System zu verstehen und dann herrausfinden an welchen Stellen sie die Architekturbeschreibung unterschiedlich verstanden haben. Da Weaknesses oftmals aus Missverständnissen entstehen kann an den identifizierten Stellen nach neuartigen Weaknesses gesucht werden, die durch eine Attack Resistance Analysis nicht gefunden werden.

3. Im Rahmen einer *Weakness Analysis* werden die Abhängigkeiten zu anderer Komponenten wie Frameworks, Betriebssystem und ihrer Einsatzumgebung untersucht. Dabei wird überprüft, ob die Annahmen, die die Software an diese Komponenten stellt auch verletzt werden können.

Technische Weaknesses werden mittels Risk Management auf Management Ebene übersetzt wodurch die Folgen einer erfolgreichen Ausnutzung einer Vulnerability in monetären Verlust des Unternehmens übersetzt werden. Durch diese Übersetzung soll die Unterstützung des Managements gewonnen werden.

2.3. Agile Modellgetriebe Entwicklung

Software wird überwiegend in einer oder mehreren General Purpose Language (GPL) geschrieben, die es ermöglicht alle Berechnungen auszudrücken, die ein Computer durchführen kann. Dabei definieren GPLs das gesamte Verhalten eines Programms in einer Form, die sich gut in Maschinensprache für eine CPU übersetzen lässt. Beim Model-Driven Development (MDD) verliert dieser GPL Programmcode seine Stellung als primäres Entwicklungsartefakt und an seine Stelle treten Modelle [Sel03].

Der hierbei verwendete Begriff des Modells lässt sich nach Stachowiak [Sta73] durch drei wesentliche Merkmale charakterisieren:

- **Abbildungsmerkmal**: Das Modell besitzt ein Original
- **Verkürzungsmerkmal**: Von Eigenschaften des Originals wird durch das Modell abstrahiert
- **Pragmatisches Merkmal**: Das Modell erfüllt einen Zweck

Ein Modell dient demnach dazu einen oder mehrere Aspekte eines Originals möglichst gut beschreiben zu können. Dies geht immer mit einer Abstraktion einher, bei der Eigenschaften, die für den zu modellierenden Aspekt nicht relevant sind weggelassen werden, sodass die Komplexität des Modells begrenzt wird. Modelle lassen sich in der Softwareentwicklung rein als Dokumentation einer Implementierung einsetzen, oder im Sinne von MDD zur Konstruktion eines Systems [Sel03]. Wenn Modelle konstruktiv eingesetzt werden besteht die Besonderheit, dass das Modell vor dem Original existiert, sodass im Konstruktionsprozess des Originals die im Modell ausgelassenen Details hinzugefügt werden müssen.

Ein solcher MDD Ansatz ist die Model Driven Architecture (MDA) [Obj14a, KWB03], welche von der Object Management Group (OMG) standardisiert wird. Diese unterteilt Modelle in verschiedene Abstraktionsstufen und sieht vor, dass abstrakte Modelle, die das zu lösende Problem beschreiben in konkretere Modelle transformiert werden, die technische Details einer Realisierung beschreiben. Die abstrakteste Stufe ist dabei das Computation Independet Model (CIM),

welches zur Erfassung der Anforderungen und Definition des Vokabulars der Problemdomäne gedacht ist, ohne dabei technische Aspekte zu beschreiben. Das Platform Independent Model (PIM) spezifiziert die fachlichen Aspekte eines Systems und erst auf der nächst konkreteren Abstraktionsstufe dem Platform Specific Model (PSM) werden erstmals technische Aspekte der Realisierung beschrieben. Das PSM entsteht aus einem PIM, indem durch automatisierte Transformationen die technischen Aspekte hinzufügt werden. Dabei wird das Ziel verfolgt, dass sich die Transformation mit geringem Aufwand wiederholen und automatisieren lassen, damit Änderungen eines CIM oder PIM leicht in ein PSM überführt werden können.

Um die Verarbeitung von Modellen automatisieren zu können, müssen diese durch einen Computer erkannt werden und gewisse syntaktische und semantische Bedingungen erfüllen. Modelle die diese Bedingungen erfüllen gehören zu einer Modellierungssprache, welche die Syntax und Semantik aller Modelle definiert, die zu dieser Sprache gehören. Der wesentliche Unterschied zwischen einer GPL und einer Modellierungssprache entsteht aus dem Verkürzungsmerkmal von Modellen, welches es erlaubt, dass eine Modellierungssprache nur einige wenige Eigenschaften eines Systems beschreibt. Somit werden unterschiedliche Sprachen eingesetzt, die jeweils besonders gut geeignet sind, um eine spezielle Sicht des Systems zu beschreiben, sodass sich aus ihrem Zusammenspiel die Beschreibung des Gesamtsystems ergibt. Die am weitesten verbreitete Modellierungssprache UML [Obj15, MLM+13] bietet für diesen Zweck eine Sammlung von Sprachen, die zusammen eine Beschreibung des Gesamtsystems erlauben. Modellierungssprachen und insbesondere die UML werden zur Spezifikation und Dokumentation in der Entwurfsphase eingesetzt um vor der Implementierung die wesentlichen Eigenschaften des zu implementierenden Systems zu definieren und von unwesentlichen Details der Implementierung zu abstrahieren.

Ein agiler Entwicklungsprozess [BBvB+01], der von einer effizienten und pragmatischen Vorgehensweise geprägt ist, in der der Quellcode die Dokumentation darstellt lässt sich durch den MDD Ansatz mit Modellierungssprachen wie der UML kombinieren [Rum12, 2.4]. In diesem Ansatz ermöglichen es Modelle Eigenschaften, die nicht direkt in GPL Code erkennbar sind zu dokumentieren, ohne das diese Dokumentation der pragmatischen agilen Methodik widerspricht, denn die Modelle definieren einen Teil der Implementierung und sind keine redundante Dokumentation. Somit ermöglicht es agile MDD auch Architekturmodelle, Datenmodelle oder andere Abstraktionen von GPL Code in der agilen Entwicklung zu verwenden.

Damit diese Modelle zur Konstruktion in einem agilen Entwicklungsprozess genutzt werden können und nicht nur als Dokumentation gesehen werden müssen sie ausführbar sein, also in Maschinensprache oder eine GPL übersetzt werden, die dann durch eine CPU ausgeführt werden kann [Rum11, Rum12, Sel03]. Diese Übersetzung muss automatisiert erfolgen, damit Änderungen an Modellen direkt zu Änderungen im implementierten System führen und die Vorteile des iterativen Vorgehens nicht durch den Einsatz von Modellen verloren geht. Eine automatische Übersetzung hat dabei noch weitere positive Implikationen für den Entwicklungsprozess. Zum einen besteht bei der manuellen Übersetzung eines Modelles durch Menschen in GPL Code die Gefahr, dass die Implementierung nicht der Spezifikation aus dem Modell entspricht. Dies kann gewollt oder ungewollt passieren, in jedem Fall entsprechen die Modelle nicht mehr der Implementierung, wodurch sie veralten und ihren Nutzen verlieren, denn um die Eigenschaften, die sie beschreiben zuverlässig zu verstehen reicht es nun nicht mehr aus die Modelle zu lesen, sondern die gesamte Implementierung muss betrachtet werden. Zum anderen ist der Aufwand

CD

```
1 package de.rwth.se.basics;
2
3 import java.util.List;
4 import java.lang.String;
5
6 classdiagram Conference {
7
8   abstract class Event {
9     public List getParticipants();
10    public void register();
11    public int price;
12  }
13
14  class Conference extends Event {
15    public boolean hasDinner;
16  }
17
18  class Workshop extends Event {
19    public boolean hasProceedings;
20  }
21
22  association Workshop <-> Conference;
23 }
```

Listing 2.3: Dieses Klassendiagramm in UML/P Notation beschreibt genau wie die graphische Version in Abbildung 2.4 die Datenstruktur einer Konferenz.

einer manuellen Übersetzung in der Regel so hoch, dass Änderungen aus Kostengründen direkt in der Implementierung durchgeführt werden und die Modelle nicht an die geänderte Implementierung angepasst werden. Eine fehlende automatische Übersetzung von Modellen in GPL Code ist somit ein Grund dafür, dass Modelle mit der Zeit vom Code abweichen und ihr Nutzen als Dokumentation verschwindet.

Um ein Modell in der modellgetrieben Entwicklung besonders gut verwenden zu können, muss es nach Selic [Sel03] folgende Eigenschaften erfüllen:

- **Abstraction**: Unnötige Details des Originals werden verborgen, damit die Essenz für den Betrachter des Modells klarer und die Komplexität des Originals beherrschbar wird.
- **Understandability**: Komplexe Sachverhalte müssen prägnant und verständlich im Modell dargestellt werden, damit der Betrachter mit geringem Aufwand den Sachverhalt verstehen kann.
- **Accuracy**: Das Modell muss sich wie das Original verhalten.
- **Predictiveness**: Mit Hilfe des Modells sollte sich eine nichttriviale Eigenschaft des Originals vorhersagen lassen.

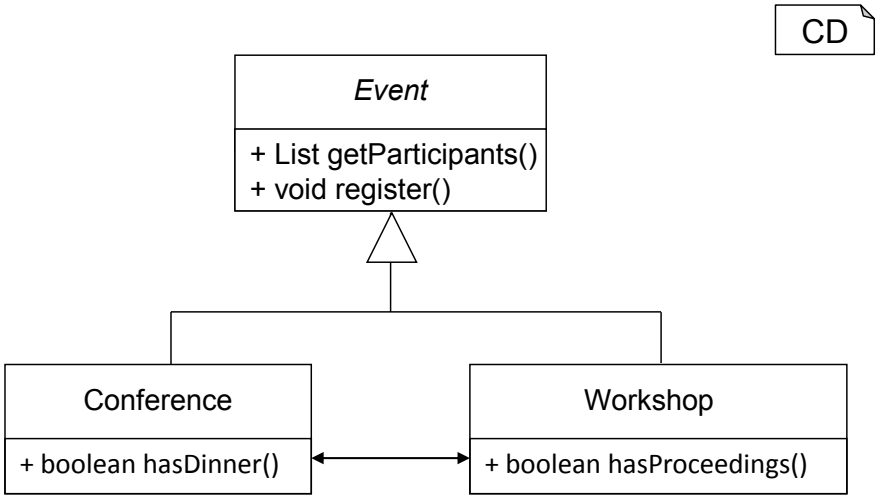


Abbildung 2.4.: Dieses Klassendiagramm in grafischer UML Darstellung beschreibt genau wie die textuelle Version in Listing 2.3 die Datenstruktur einer Konferenz.

- **Inexpensive**: Die Erstellung und Analyse des Modells muss wesentlich günstiger sein, als die Analyse direkt auf dem Original durchzuführen.

Graphische Modellierungssprachen wie die UML eignen sich gut zur Kommunikation mit Menschen. Um diese in der MDD zu verwenden muss allerdings die Syntax und Semantik jedes Modellelements exakt definiert sein, damit *accuracy* und *predictiveness* des Modells im Entwicklungsprozess sichergestellt ist. Für den Einsatz in der agilen MDD ist es außerdem notwendig, dass Entwickler Modelle genauso wie GPL Code schnell und mit wenig Aufwand ändern können. Die textuelle Notation von GPLs hat sich zu diesem Zweck bewehrt. Außerdem ist eine präzise Syntax in textuellen Modellierungssprachen allein dadurch notwendig und gegeben, dass Modelle vor ihrer Verarbeitung durch einen Parser erkannt werden müssen. Eine eindeutige Übersetzung von syntaktisch validen Modellen einer Modellierungssprache in ausführbaren GPL Code definiert dann eine Semantik für diese Modellierungssprache. Um die präzise Semantik einer Sprache zu definieren und sie damit im MDD verwendbar zu machen wird daher im Folgenden immer die textuelle Syntax einer Sprache betrachtet. Daher wird hier auch die textuelle Notation für UML/P Klassendiagramme und Statecharts [Rum11, Sch12] anstelle der graphischen UML Notation der OMG verwendet.

Die UML/P Notation für Klassendiagramme lehnt sich dabei an GPLs wie Java an, sodass die Definition einer abstrakten Klasse in Zeile 8 von Listing 2.3 mit der Syntax von Java identisch ist. Genauso verhält es sich mit der Vererbung zwischen Klassen, die durch `extends` angegeben wird. Assoziationen zwischen Klassen werden mit dem Schlüsselwort `association` definiert, können uni- oder wie in Zeile 22 bidirektional sein und auch Kardinalitäten besitzen. Abbildung 2.4 zeigt das Klassendiagramm aus Listing 2.3 in graphischer UML Notation. Die beiden Klassendiagramme modellieren das Datenmodell einer Konferenz, bei der es unterschiedliche Veranstaltungen gibt, zu denen man sich anmelden kann und deren Teilnahmegebühr unterschiedlich sein kann.

SC

```
1 package de.rwth.se.basics;
2
3 statechart Registration {
4   initial state Unregistered;
5
6   Unregistered -> Registered : / {
7     submitForm("Max Mustermann");
8   }
9
10  state Registered {
11    initial state Unpayed;
12    state Payed {
13      entry / { addTurnover(); }
14    }
15    Unpayed -> Payed;
16  }
17 }
```

Listing 2.5: Dieses Statechart in UML/P Notation beschreibt genau wie die graphische Version in Abbildung 2.6 das Verhalten des Registrierungsprozess einer Konferenz.

Ebenfalls für diese Konferenz zeigt Abbildung 2.6 die graphische UML Notation eines Statecharts, welches das Verhalten des Registrierungsprozesses beschreibt. In Listing 2.5 ist dieses Statechart in der textuellen UML/P Notation dargestellt. Dort wird in Zeile 4 der Startzustand durch `initial state` definiert, aus dem über die in Zeile 6-8 definierte Transition der hierarchische Zustand `Registered` (Zeile 10-16) erreicht werden kann. Die Transition definiert in den geschweiften Klammern eine Aktion, die das System ausführt, wenn die Transition ausgelöst wird. Hier wird die Aktion in Java definiert, sodass die Methode `submitForm("Max Mustermann");` ausgeführt wird. Der hierarchischen Zustands `Registered` kapselt das in ihm definierte Verhalten, sodass durch die Transition von `Unregistered` nach `Registered` nicht festgelegt ist, welche inneren Zustände von `Registered` als nächstes angenommen werden. Daher wird innerhalb des hierarchischen Zustands zuerst der als Startzustand definierte Zustand der angenommen, wenn der Zustand `Registered` angenommen wird. Der in Zeile 12-14 definierte innere Zustand `Payed` besitzt eine `entry` Aktion, die ausgeführt wird, wenn dieser angenommen wird, welche ebenfalls in Java definiert ist.

2.3.1. Modellverarbeitung mit MontiCore

Damit agile MDD in Projekten angewendet werden kann bedarf es einer Werkzeuginfrastruktur zur Modellverarbeitung, die sich leicht und pragmatisch an die Bedürfnisse des Projekts anpassen lässt. Zur Erstellung von Werkzeugen zur Verarbeitung von textuellen Sprachen gibt es verschiedene Language Workbenches [EvdSV+13], die jeweils unterschiedliche Konzepte zur Sprachverarbeitung und -Definition unterstützen. Eine Möglichkeit der Sprachdefinition besteht darin eine Grammatik anzugeben, welche zum erkennen einer Modellierungssprache

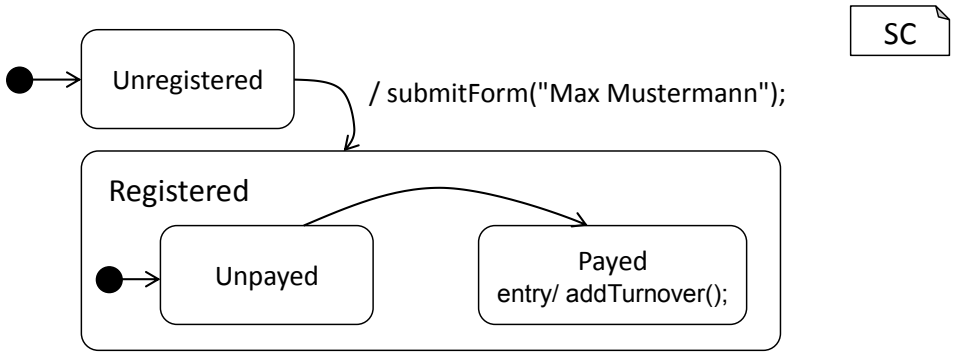


Abbildung 2.6.: Dieses Statechart in grafischer UML Darstellung beschreibt genau wie die textuelle Version in Listing 2.5 das Verhalten des Registrierungsprozess einer Konferenz.

durch einen aus der Grammatik generierten Parser verwendet wird. Exemplarisch kann man hier Xtext [EB10] nennen, welches von der Itemis AG als Teil des Eclipse Projekts entwickelt wird und Sprachverarbeitungswerkzeuge bereitstellt, die auf der Eclipse IDE basieren. In dieser Arbeit wird MontiCore [KRV10, Kra10] zur Modellverarbeitung eingesetzt, welches speziell auf die Anforderungen der agilen MDD ausgerichtet ist.

Abbildung 2.7 gibt einen Überblick über die wichtigsten Komponenten (Parser, Transformation Engine, Template Engine), Artefakte (Modell, MontiCore Grammatik, Transformation, Freemarker Template, Code) und Datenstrukturen (AST) sowie deren Zusammenhang bei der Nutzung des MontiCore Frameworks. MontiCore hat das Ziel die agile und kompositionale Entwicklung von Modellierungssprachen zu unterstützen, sodass die Modelle konstruktiv in der agilen MDD verwendet werden können. Damit ein Modell verarbeitet werden kann muss zuvor die Modellierungssprache definiert werden, zu der dieses Modell gehört. Eine solche Sprache wird über eine Kontextfreie Grammatik definiert und in der Notation einer MontiCore Grammatik (MCG) aufgeschrieben, die neben Produktionen die die Sprache definieren auch Mechanismen zur Sprachkomposition unterstützt. Aus der vom Sprachentwickler erstellten MontiCore Grammatik generiert MontiCore einen Parser, Java Klassen, die den AST von Modellen der Sprache darstellen und eine Transformation Engine. Wird ein Modell mit MontiCore verarbeitet, so liest zuerst der Parser das Modell ein und instantiiert die generierten Klassen, sodass die resultierende Objektstruktur der AST des eingelesenen Modells ist. Dieser AST wird als Repräsentation des Modells bei der weiteren Verarbeitung innerhalb von MontiCore verwendet. Die aus der MCG abgeleitete Transformation Engine erlaubt es den AST mit Transformationen zu verändern. Diese Transformationen werden in einer domänenspezifischen Transformationssprache beschrieben [Wei12, HRW15b], welche die Syntax der durch die MCG definierten Sprache verwendet, wodurch diese Transformationen ebenfalls das in den Modellen verwendete Vokabular nutzen.

MontiCore verwendet templatebasierte Codegenerierung, um einen AST durch die Template Engine in Code zu überführen [Sch12]. Ein Template beschreibt dabei den zu generierenden Code exemplarisch unter Verwendung einer Templatesprache, die sich in den zu generierenden Code einpasst und Kontrollstrukturen zur Verfügung stellt, mit denen beispielsweise Codeblöcke unter

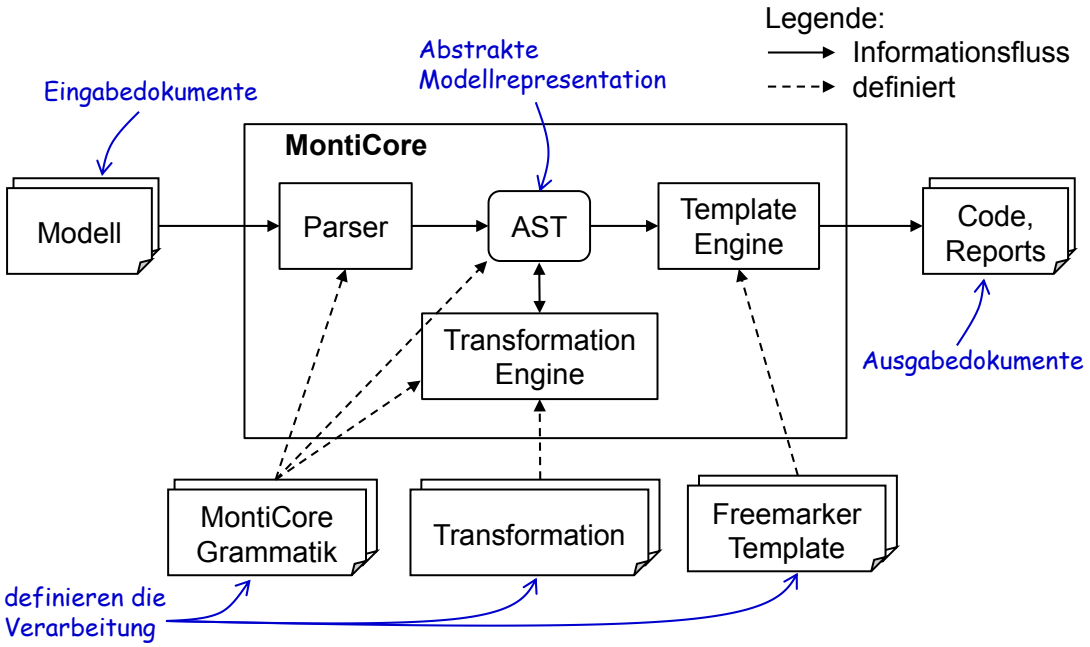


Abbildung 2.7.: Überblick über die Komponenten, Artefakte und Datenstrukturen von MontiCore.

bestimmten Bedingungen oder mehrfach hintereinander generiert werden können. In MontiCore wird Freemarker [Fre15] als Templatesprache eingesetzt, die ein Generatorentwickler verwendet, um den Code zu beschreiben, der aus einem Modell generiert wird. MontiCore bietet dem Generatorentwickler weitere Hilfsmittel um komplexere Operationen, die über die Ausgabe von Code hinausgehen in Java durchzuführen, um die Komplexität von Templates gering zu halten.

Somit lassen sich Modelle in ein ausführbares Systemen übersetzen, welches die im Modell spezifizierten Eigenschaften besitzt. Da ein Modell von technischen Details abstrahiert, die für den Modellzweck unerheblich sind, müssen diese durch Transformationen oder bei der Implementierung eines Generators wieder hinzugefügt werden. Daher trifft der Generatorentwickler Annahmen über Eigenschaften, die nicht im Modell spezifiziert sind beispielsweise ob das System als Web-System, Desktop Anwendung in Java oder Python realisiert werden soll. Unterschiedliche Generatoren generieren daher aus dem selben Modell verschiedene Implementierungen.

Neben den Hilfsmitteln zur Modellverarbeitung bietet MontiCore auch Mechanismen zur effizienten Entwicklung, Anpassung und Erweiterung von Modellierungssprachen durch Sprachkomposition. Dies wird durch Grammatikerweiterung ermöglicht, die das Konzept der Vererbung im Sinne der Objektorientierung auf Gramatiken überträgt [Kra10]. Bei der Erweiterung werden zwei Fälle unterschieden. Zum einen lassen sich Produktionen einer Grammatik durch das Schlüsselwort `extends` erweitern und zum anderen ist auch eine Erweiterung von Grammatiken über das Schlüsselwort `extends` möglich.

In Listing 2.8 in Zeile 2 wird die Produktion von A, die in Zeile 1 definiert ist um eine weitere Alternative erweitert. Diese Erweiterung ist inspiriert aus der Objektorientierten Programmierung,

```
1 A = "hello";
2 B extends A = "world";
3 C = A;
```
MCG

Listing 2.8: Eine MontiCore Grammatik mit Grammatikvererbung.

```
1 interface I;
2 A = I "...";
3 B implements I = "...1";
4 C implements I = "...2";
```
MCG

Listing 2.9: Eine MontiCore Grammatik, die ein `interface` Nichtterminal verwendet.

wo eine Klasse A nach ihrer Erstellung durch eine Klasse B erweitert werden kann. Dieser Mechanismus in MontiCore Grammatiken erlaubt es bestehende Grammatiken wiederzuverwenden, zu erweitern und anzupassen ohne die zu erweiternde Grammatik zu verändern. Somit können bei der Entwicklung einer neuen Modellierungssprache bestehende MontiCore Grammatiken, die beispielsweise grundlegende Namen und Whitespaces definieren wiederverwendet werden. Genauso lassen sich komplette Modellierungssprachen durch die Erweiterung weniger Produktionen um neue Aspekte erweitern. Um die Erweiterung einer Sprache explizit vorzusehen bietet MontiCore die Möglichkeit Nichtterminale als `interface` zu deklarieren, wie in Zeile 1 in Listing 2.9 zu sehen. Diese können als normales Nichtterminal verwendet werden (Zeile 2), auch wenn noch keine Produktion für dieses Nichtterminal definiert wurde.

```
1 I = B | C;
2 A = I "...";
3 B = "...1";
4 C = "...2";
```
MCG

Listing 2.10: Diese MontiCore Grammatik erkennt die selbe Sprache wie die MontiCore Grammatik in Listing 2.9.

Produktionen für dieses Nichtterminal werden erst zu einem späteren Zeitpunkt, wie in Zeile 3 und 4 zu sehen, durch das Schlüsselwort `implements` hinzugefügt. Jedes Nichtterminal, dass das Interface Nichtterminal `I` implementiert ergibt eine Alternative in der Produktion von `I`. Somit definiert die in Listing 2.10 dargestellte MontiCore Grammatik die gleiche konkrete Syntax wie die in Listing 2.9.

Der Nutzen dieser Vererbungsmechanismen entsteht, wenn sie genutzt werden, um eine bestehende Grammatik wiederzuverwenden und sie dabei zu erweitern und anzupassen. Listing 2.11 zeigt die Grammatik `SuperLang` und die Grammatik `SubLang`, welche `SuperLang` erweitert. Durch das `interface` Nichtterminal in Zeile 3 bietet `SuperLang` explizit die Möglichkeit zur Erweiterung, welche `SubLang` nutzt.

```
grammar SuperLang {
  A = "hello";
  interface B;
  C = [ A | B ];
}

grammar SubLang extends SuperLang {
  NewB implements B = "world";
}
```

MCG

Listing 2.11: Eine MontiCore Grammatik erweitert eine andere Grammatik.

2.3.2. Architekturbeschreibungssprachen

Es gibt verschiedene Definitionen, die beschreiben was die Software-Architektur eines Systems ist [PW92]. Hier wird die Definition von Taylor [TMD09, S. 58] verwendet:

Definition 2.3.1 (Software-Architektur) *Die Architektur eines Systware-Systems ist die Menge aller grundlegenden Entwurfsentscheidungen, die über dieses System getroffen wurden.*

Der ISO Standard 42010 [ISO11b] unterscheidet explizit die Architektur von der Architekturbeschreibung, da zwar jedes System eine Architektur besitzt, diese jedoch nicht notwendigerweise auch in einer Architekturbeschreibung dokumentiert ist. Die Architekturbeschreibung ist demnach ein Modell der Architektur, welches in einer Architecture Description Language (ADL) formuliert werden kann [MT00] Da beim Systementwurf grundlegend verschiedene Aspekte wie die physische Verteilung, die logische Struktur oder das grundsätzliche Verhalten eines Systems festgelegt werden, sind für diese Aspekte oder Concerns unterschiedliche Sichten auf die Architektur notwendig, die jeweils nur einen Concern beschreiben, diesen aber möglichst präzise und anschaulich darstellen. Durch die Modellierung in einer ADL wird die Architektur explizit sichtbar, sodass sie diskutiert, bewertet und in der MDD auch während der Entwicklung verändert werden kann. Im Allgemeinen kann eine ADL mehrere Modellierungssprachen zusammenfassen, die unterschiedliche Sichten auf die Architektur beschreiben, sodass die ADL die gesamte Architektur beschreibt. Im Folgenden meint der Begriff ADL jedoch eine Modellierungssprache, die nur eine Sicht auf die strukturelle Architektur beschreibt.

Um die Struktur eines verteilten Systems zu beschreiben werden Component & Connector ADLs (C&C ADLs) verwendet, die ein System in Komponenten (Teile) unterteilen und mit Konnektoren die Interaktionsmöglichkeiten, also die möglichen Kontroll- und Datenflüsse, zwischen Komponenten darstellen [TMD09, S. 70]. Dabei wird das Prinzip der Kapselung und Abstraktion angewandt, um die Komplexität eines Gesamtsystems zu beherrschen, indem es in einzelne Komponenten zerteilt wird.

Eine Komponente in einer Software-Architektur kennzeichnet dabei, dass sie:

1. einen Teil der Funktion eines Gesamtsystem kapselt,
2. der Zugang zu den Funktionen einer Komponente nur über ein explizit definiertes Interface möglich ist, und

3. die zur Ausführung der Funktionen notwendigen Abhängigkeiten ebenfalls explizit definiert sind [MT00, TMD09, S. 69].

Komponenten sind somit voneinander entkoppelt und haben keine weiteren als die über Interfaces explizit definierten Anforderungen an andere Komponenten. Durch die Kapselung in Komponenten wird es möglich die wesentlichen Elemente eines Systems zu beschreiben, ohne alle Details über ihre Funktionsweise zu betrachten.

Eine Komponente kann entweder atomar sein, oder sie besteht selbst wieder aus Komponenten, die mit Konnektoren verbunden sind, sodass man von einer komponierten Komponente spricht. Durch diese Dekomposition einer komponierten Komponente in mehrere Subkomponenten wird die Funktionalität des Systems dekomponiert, sodass die Subkomponenten und deren Konnektoren zusammen die Funktion der komponierten Komponente definieren. Diese Subkomponenten werden auch als Architectural Configuration bezeichnet [MT00] und die Dekomposition einer Komponente in Subkomponenten als Refinement [PR99].

Es gibt zahlreiche C&C ADLs, die unterschiedliche Notationen und Konzepte verwenden, um Komponenten und Konnektoren darzustellen und zu präzisieren. In der Praxis wird als häufigste C&C-Architektursprache die Architecture Analysis & Design Language (AADL) [FGH06] eingesetzt, wie eine Umfrage unter 40 Unternehmen ergab [MLM+13]. Hier wird die ADL MontiArc [HRR12, Hab16] verwendet, die in ihren Grundkonzepten zur AADL kompatibel ist. Anhand eines einfachen Zufahrtskontrollsystem eines Parkhauses zeigt Abbildung 2.12 die graphische und Listing 2.13 die textuelle Syntax von MontiArc. Das System besteht aus einer komponierten Komponente (Zeile 3-13), die aus zwei atomaren Komponenten (Zeile 5-10) zusammengesetzt ist, deren Ports durch einen Konnektor (Zeile 12) verbunden sind.

MontiArc nutzt FOCUS [BS01] als formales Fundament, sodass MontiArc Komponenten und Konnektoren stromverarbeitenden Funktionen in FOCUS entsprechen, die asynchron über gerichtete Kanäle kommunizieren. Durch die Semantik von FOCUS lässt sich das Verhalten von MontiArc Modellen simulieren und zusammen mit der Verhaltensspezifikation von atomaren Komponenten ausführen. Dadurch lässt sich MontiArc in der agilen MDD auch einsetzen, wenn noch nicht alle Komponenten implementiert und vollständig spezifiziert sind.

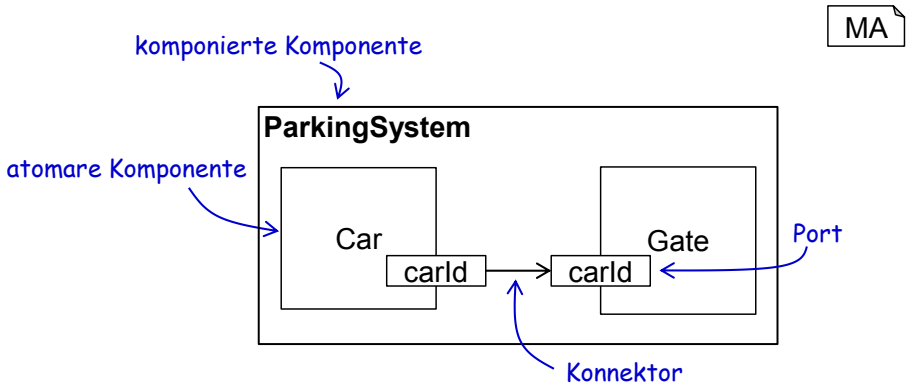


Abbildung 2.12.: Dieses graphische MontiArc Modell beschreibt genauso wie Listing 2.13 die Zufahrtskontrolle eines Parkhauses.

MA

```
1  package de.rwth.se.basics;
2
3  component ParkingSystem {
4
5    component Car {
6      port out CarId;
7    }
8    component Gate {
9      port in CarId;
10   }
11
12   connect car.carId -> gate.carId;
13 }
```

Listing 2.13: Dieses textuelle MontiArc Modell beschreibt genauso wie Abbildung 2.12 die Zufahrtskontrolle eines Parkhauses.

Kapitel 3.

Anforderungen an Modelle und Werkzeuge

In diesem Kapitel werden die Herausforderungen bei der Umsetzung von Software Security in der agilen Modellgetriebenen Entwicklung dargestellt und Anforderungen an die dabei verwendeten Modelle und Werkzeuge definiert.

Fehler in einer Software nach der Auslieferung zu beheben kostet wesentlich mehr als sie bereits während der Entwicklung zu beheben [BB01]. Dies kann durch eine permanente Beta-Phase abgedämpft werden, in der Software mit schlechten Qualitätseigenschaften ausgeliefert wird und erst mit der Zeit durch Fehlermeldungen ihrer Nutzer den Feinschliff erhält. Solche Beta-Software eignet sich jedoch nicht für den Einsatz in Unternehmen, wo sie Geschäftsprozesse steuert und wertvolle Daten verarbeitet, da Security Vulnerabilities in einer solchen Umgebung immer das Risk erhöhen, dass diese Systeme kompromittiert werden. Daher müssen Fehler und insbesondere Security Weaknesses möglichst früh nach ihrer Entstehung im Entwicklungsprozess behoben werden.

Aus diesem Grund priorisieren McGraw und der SDL von Microsoft Aktivitäten, die in den frühen Entwicklungsphasen liegen. Für die Anforderungserhebung existieren Techniken und Werkzeuge, die es ermöglichen Security-Anforderungen von Kunden z.B. in Form von Misuse Cases [Ale03] oder AttackTrees [Sch99] zu dokumentieren. Diese Anforderungen beschreiben Szenarios, die während des Betriebes eins Systems nicht eintreten sollen, allerdings ohne anzugeben, wie dies erreicht werden soll. Und wenn Schutzmaßnahmen explizit genannt werden bleibt offen, wie diese implementiert werden müssen, damit sie nicht von einem Angreifer umgangen werden können.

Alle möglichen (technischen) Angriffe, die erst durch Weaknesses im Design oder der Implementierung ermöglicht werden individuell für jedes System in den Anforderungen, beispielsweise in Form von Misuse Cases, zu definieren bedeutet einen erheblichen Aufwand. Außerdem führt dies dazu, dass für jeden Use Case mehrere Misuse Cases entstehen, sodass die fachlichen Anforderungen in den Hintergrund gedrängt werden. Da diese Anforderungen nur festlegen, dass keine Weaknesses entstehen dürfen, gleichzeitig jedoch offen lassen wie eine Implementierung realisiert werden soll, lässt sich das Software Security-Problem alleine mit Anforderungen nicht lösen.

Die in Abschnitt 2.1.4 beschriebenen grundlegenden Designprinzipien geben Hinweise darauf, wie sichere Systeme entworfen werden sollen. Da die Prinzipien jedoch allgemein formuliert sind, ist eine direkte Anwendung auf die im Softwareentwicklungsprozess zu treffenden Entscheidungen schwierig. Auch die Sammlungen von bekannten Vulnerabilities (CVE) [Nat15], Weaknesses (CWE) [MIT15b] und Angriffen (CAPEC) [MIT15a] dienen Entwicklern eher als

Schulungsmaterial, welches sich jedoch nicht direkt konstruktiv in einem Entwicklungsprozess verwenden lässt. Wie McGraw et al. [McG03], [McG06, S. 319] und Wing [Win03] aufzeigen fehlen Entwicklern bei Entwurf und Implementierung Methoden und Werkzeuge, die sie dabei unterstützen, Security Weaknesses zu verhindern bzw. zu erkennen und zu beheben. Insbesondere zeigt McGraw folgende Fragen auf, die in dieser Arbeit behandelt werden:

1. Wie kann verhindert werden, dass Flaws und Bugs in Systeme eingebaut werden?
2. Wie müssen Systeme entworfen werden, damit sie Angriffe tolerieren und ihre Aufgabe trotz eines Angriff weiter ausführen?
3. Wie können Security-Architekturen automatisiert auf Flaws analysiert werden?

Für Privacy gelten die zuvor dargestellten Beobachtungen analog [McG06, S. 319]. Es existieren grundlegende Prinzipien, die beim Entwurf eines Systems eingehalten werden sollen [Cav16, SC09, Pea09], allerdings fehlen Konzepte, wie Privacy in den Entwicklungsprozess integriert werden kann und Werkzeuge, die Entwickler dabei unterstützen.

Im Folgenden werden die wesentlichen Anforderungen an Methoden und Werkzeuge sowie die im Entwicklungsprozess verwendeten Modelle dargestellt, die Entwickler dabei unterstützen sollen Security und Privacy bereits im Entwurf mit in ein System einzubauen:

A1 **Agile modellgetriebene Entwicklung**
Die agile modellgetriebene Entwicklung verspricht durch den Einsatz von Modellen Eigenschaften, die nur schwer in GPL Code darzustellen sind, wie Security und Privacy, auch bei sich ändernden Anforderungen beherrschbar zu machen. Daher sollen Security und Privacy in einen agilen modellgetriebenen Entwicklungsprozess integriert werden und so Entwickler bereits beim Entwurf unterstützt werden sichere Systeme zu konzipieren.

A2 **Security und Privacy für alle Systeme**
Für jedes System, mit dem in irgendeiner Form Geld verdient wird oder kritische Prozesse gesteuert werden sind Security und Privacy notwendige Eigenschaften. Methoden und Werkzeuge müssen daher auf die breite Masse an Systemen zugeschnitten sein und nicht nur auf besonders kritische Anwendungen, bei denen erhebliche zusätzliche Aufwände in die Vermeidung von Vulnerabilities investiert werden. Die Integration von Security und Privacy darf daher nicht mit erheblichem Zusatzaufwand verbunden sein.

A3 **Don't make me think about security and privacy**
Entwickler wollen sich auf die Realisierung der eigentlichen Businesslogik eines Systems konzentrieren, da sie eine konstruktive Sichtweise haben und daran interessiert sind funktionale Anforderungen zu erfüllen. Sie sind keine Security-und Privacy-Experten, die eine destruktive Sichtweise haben [Ale03], sollen aber in der Lage sein Security und Privacy zu einem hohen Maße ohne Unterstützung eines Experten zu implementieren.

A4 **Fehlervermeidung durch Complete Mediation**
Viele Vulnerabilities entstehen, da existierende Technologien wie beispielsweise HTTP verwendet werden, die ursprünglich keine Security-Mechanismen hatten. Diese Mechanismen wurden nachträglich hinzugefügt und deren Nutzung bzw. Aktivierung wurde

aus Gründen der Kompatibilität mit älteren Versionen dem Entwickler überlassen. Damit entsteht eine Situation, die als Insecurity-by-default beschreiben werden kann, in der die verwendete Technologie standardmäßig keine Sicherheitsmechanismen einsetzt, oder diese optional sind, sodass ein Angreifer sie deaktivieren kann. Da Entwickler zur Realisierung von funktionalen Anforderungen nicht dazu angehalten werden Security-Schutzmaßnahmen zu implementieren, entstehen i.d.R. Systeme, die zwar alle funktionalen Anforderungen erfüllen, jedoch viele Vulnerabilities enthalten. Damit eine Insecurity-by-default Technologie eingesetzt werden kann, um sichere Systeme zu realisieren, müssen die Entwickler für die richtige Verwendung dieser Technologie geschult werden, wie es OWASP für Web-Technologien tut. Allerdings unterlaufen selbst den bestgeschultesten Entwicklern Fehler, besonders wenn sie ein System mit komplexer Insecurity-by-default Technologie realisieren müssen. Im Kontrast dazu erlaubt es eine Technologie, die dem Security-Designprinzip Complete Mediation folgt gar nicht erst unsichere Systeme zu erstellen. Entwickler müssen sich hierbei natürlich an gewisse Regeln halten, sodass die von der Technologie eingeführten Schutzmaßnahmen nicht durch Weaknesses umgehbar gemacht werden, die im Laufe der Systementwicklung eingeführt werden. Allerdings müssen dies Regeln wesentlich einfacher zu befolgen sein als alle Richtlinien zur richtigen Benutzung von Insecurity-by-default Technologie zu beachten.

Die verwendeten Modellierungssprachen und Werkzeuge sollen verhindern, dass unsichere Systeme entstehen und somit das Security-Designprinzip Complete Mediation umsetzen.

A5 **Konkrete Unterstützung für den Entwicklungsprozess**
Die entwickelten Methoden und Werkzeuge sollen sich in vorhandene allgemeine Prozesse und Prinzipien, z.B. ISO 27001 [ISO13] oder Privacy-by-Design [Cav16], integrieren lassen, selber jedoch konkretere Unterstützung im Softwareentwicklungsprozess bieten als diese allgemeinen Rahmenwerke.

A6 **Nachweisbarkeit**
Nichtfunktionale Eigenschaften wie Security und Privacy sind durch den Nutzer nur in geringem Ausmaß erfahrbar. Gleichzeitig sind sie für die Akzeptanz eines Systems wichtig. Die Nachweisbarkeit dieser Eigenschaften soll vereinfacht, automatisiert und wo möglich durch Modelle so abstrahiert werden, dass Nutzer selbst nachvollziehen können wie diese Eigenschaften von einem System erbracht werden.

In den folgenden Kapiteln werden mehrere Ansätze vorgestellt, die diese Anforderungen berücksichtigen und verfeinern, um verschiedene Aspekte von Security und Privacy in einen agilen modellgetriebenen Entwicklungsprozess zu integrieren.

Kapitel 4.

Security-Architekturmodellierung mit MontiSecArc

> No one team designs all the layers, ever, because the story of computer science is that of nesting layers all the way down to the analog (which occasionally does bite us, hello Rowhammer).
>
> Dan Kaminski auf der LangSec Mailingliste

In diesem Kapitel wird die Security-Architekturbeschreibungssprache MontiSecArc vorgestellt. Dazu wird zunächst der Zusammenhang mit den folgenden Kapiteln dargestellt, in denen Analysen und Codegeneratoren für MontiSecArc beschreiben werden. Im Anschluss werden zunächst die im vorherigen Kapitel beschriebenen Anforderungen im Kontext der Architekturbeschreibung betrachtet bevor die Modellelemente der Sprache sowie deren Syntax und Semantik beschrieben werden.

Da Fehler in der Softwareentwicklung annähernd zu gleichen Teilen im Entwurf und in der Implementierung entstehen [Gal04] und sich die Konsequenzen eines Flaws, in mehreren Teilen der Implementierung auswirken ist es zur Reduktion der Fehlerbehebungskosten sinnvoll bereits während des Entwurfs Weaknesses zu identifizieren und beheben. Dazu bedarf es einer Modellierungssprache, die die wesentlichen Security-Eigenschaften eines Systems, also die Security-Architektur, präzise beschreiben kann. Damit diese Sprache im Entwurf eines agilen modellgetriebenen Softwareentwicklungsprozess genutzt werden kann (Anforderung A1) muss sie die iterative Vorgehensweise unterstützen und konstruktiv genutzt werden können [Rum12]. Wie in Kapitel 2.3.2 beschrieben existiert mit MontiArc [HRR12, Hab16] bereits eine Architekturbeschreibungssprache für genau diesen Einsatzzweck. Daher werden die in MontiArc vorgesehenen Erweiterungsmöglichkeiten genutzt, um der Sprache Security-Eigenschaften hinzuzufügen und

hierdurch die Security-Architekturbeschreibungssprache MontiSecArc zu definieren. Sind die Security-Eigenschaften einmal in einem Modell festgehalten, lassen sie sich automatisiert und reproduzierbar auf Vollständigkeit überprüfen, und so bekannte Flaws identifizieren, wie im nächsten Kapitel 5 beschrieben wird.

Ziel von MontiSecArc ist es Aussagen über Security-Eigenschaften in der Architektur zu ermöglichen, die durch das Zusammenspiel einzelner Schutzmaßnahmen, die im System implementiert sind entstehen. Damit wird die Grundlage zum Nachweis (Anforderung A6) dieser Eigenschaften geschaffen, der nur dann eine gehaltvolle Aussage über die tatsächliche Systemarchitektur ermöglicht, wenn die Modellelemente syntaktisch und semantisch möglichst genau definiert sind und das entwickelte System auch tatsächlich diese Semantik umsetzt. Die Verbindung zwischen einem MontiSecArc-Modell und dem implementierten System wird entsprechend der Agilen MDD und wie in Kapitel 6 beschrieben mittels Codegeneratoren realisiert.

Unter der Voraussetzung, dass durch die Codegenerierung keine systematischen Bugs, also Flaws hinzugefügt werden ist es ausreichend MontiSecArc-Modelle auf Flaws zu untersuchen. Damit ist es nicht notwendig im gesamten GPL Code der Implementierung nach Flaws zu suchen, wenn diese bereits auf Modellebene untersucht werden. Bei der Wahl der Modellelemente gilt es jedoch zu beachten, dass diese konstruktive für Entwickler nutzbar sein müssen (Anforderung A3).

4.1. Modellelemente

Ausgangspunkt für die Entwicklung der Sprache MontiSecArc ist die Component & Connector Architekturbeschreibungssprache MontiArc, welche es erlaubt die Dekomposition eines Systems in Komponenten zu beschreiben. Dabei stehen die funktionalen Eigenschaften des Systems im Vordergrund und Security wird nicht modelliert. Um Security-Eigenschaften zu identifizieren und daraus Modellelemente von MontiSecArc zu entwerfen wurden bekannte Angriffstechniken auf IT Systeme herangezogen. Bei diesem Ansatz wurde iterativ für jeden Angriff geprüft, ob die bisher identifizierten Modellelemente eine Aussage darüber erlauben, ob der Angriff möglich ist ober nicht. Dabei wird ausgenutzt, dass jeder Angriff darauf abzielt eine Security-Eigenschaft zu verletzten und im Bereich Software Security zwar wenig Literatur über Security-Eigenschaften, jedoch viel über Angriffe vorhanden ist. So wurden Angriffe, die im OWASP Testing Guide [Ope15c], den CWE [MIT15b] und Common Attack Pattern [MIT15a] beschrieben sind verwendet. Die identifizierten Eigenschaften und daraus entwickelten Modellelemente erhalten durch dieses Vorgehen einen konstruktiven Blickwinkel, sodass ein MontiSecArc-Modell beschreibt, wie eine Component & Connector-Architektur durch Gegenmaßnahmen und Schutzfunktionen Angriffe abwehrt. Zugleich zielt dieses Vorgehen darauf ab möglichst semantisch gehaltvolle Modellelemente zu entwickeln, Redundanzen zu vermeiden und damit mit wenigen Modellelementen viele Angriffe und damit Security-Eigenschaften abzudecken.

Dieser Prozess wurde im Rahmen der Arbeit von Markus Paff [Paf15] durchlaufen und die entworfenen Modellelemente von Sandra Hicks [Hic15] mit 60 Security Design Pattern und Design Principles verglichen. Durch diesen Vergleich wurden zwei Pattern identifiziert, die zu einer Erweiterung von MontiSecArc um physische Schutzmaßnahmen führten, die von Philipp Rainisch [Rai15] genauer untersucht wurden.

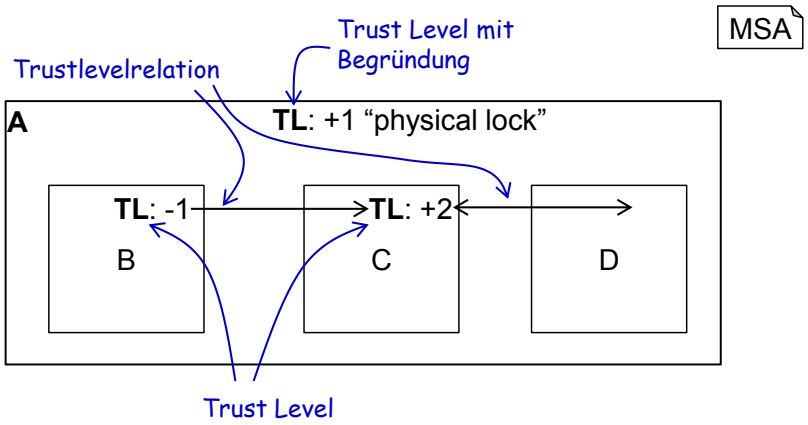


Abbildung 4.1.: Grafische Darstellung von Trustlevel und Trustlevelrelation.

Per Definition können in der Architektur nur Flaws und keine Bugs enthalten sein, da Bugs erst in der Implementierung entstehen. Daher führen Angriffe, die ausschließlich Bugs ausnutzen, wie die Klasse der Injection Angriffe (siehe Kapitel 8), nicht zu neuen Security-Eigenschaften die in der Architektur repräsentiert werden können. Durch den agilen MDD Ansatz ist es jedoch möglich manche Bugs zu vermeiden, indem eine Implementierung aus dem Architekturmodell abgeleitet wird, sodass bei korrekter Implementierung des Generators diese Bugs vermieden werden.

Im MontiArc Typsystem wird durch eine Komponente ein Typ definiert, welcher die Eigenschaften und Ports der Komponente festlegt. Um Komponenten in einer Architekturbeschreibung zu verwenden werden diese instantiiert, sodass die Instanzen die Eigenschaften und Ports der Komponente haben. Bei der Definition einer Komponente wird nun gleichzeitig eine Instanz unter dem Namen der Komponente instantiiert. Die durch MontiSecArc hinzugefügten Modellelemente werden möglichst bei der Definition von Komponenten verwendet, sodass alle Instanzen einer Komponente die gleichen Security-Eigenschaften haben. Dies erleichtert das Verständnis des Modells, da durch eine Instanziierung die Security-Eigenschaften einer Komponente nicht verändert werden. Es lassen sich jedoch nicht alle Security-Eigenschaften in die Komponentendefinition integrieren, sodass Eigenschaften, die Beziehungen zwischen unterschiedlichen Modellelementen betreffen, für Instanzen definiert werden. Für die Realisierung von MontiSecArc mit MontiCore bedeutet dies, dass alle Modellelemente von MontiArc auch in MontiSecArc genutzt werden können, da deren Syntax und Semantik erhalten bleibt und es sich um eine reine Erweiterung der Sprache handelt.

Im Folgenden werden die Modellelemente von MontiSecArc vorgestellt. Dabei wird deren Semantik über ein Mapping von Modellelementen auf bekannte Konzepte und Eigenschaften wie Security-Mechanismen und Pattern beschrieben und jeweils eine grafische und textuelle Darstellung angegeben, wobei die Textuelle präziser ist und die Graphische zur Übersicht dient.

```
1 package de.rwth.se.secarc.syntax;                    MSA
2
3 component A {
4   trustlevel +1 "physical lock";
5   trustlevelrelation b < d;
6   trustlevelrelation c > b;
7   trustlevelrelation c = d;
8
9   component B b {
10    trustlevel -1;
11  }
12
13  component C c {
14    trustlevel +2;
15  }
16
17  component D d {
18    trustlevel +2;
19  }
20 }
```

Listing 4.2: Textuelle Syntax von Trustlevel und Trustlevelrelation.

4.1.1. Trust Level

Um den Einfluss, den ein Angreifer auf Komponenten einer Architektur hat zu modellieren und somit Komponenten, die durch einen Angreifer beeinflusst werden können von denen abzugrenzen, die er nicht beeinflussen kann wird in MontiSecArc für Komponenten ein *Trust Level* eingeführt. Mit einem positiven Trust Level beschreiben Entwickler (physische) Schutzmaßnahmen, die einen Angreifer daran hindern die Berechnungen und Kommunikation innerhalb einer Komponente zu beeinflussen. Das Trust Level abstrahiert von einzelnen Schutzmaßnahmen wie verschlossenen Türen, Wänden und Videoüberwachung, damit sich das Modell auf das zu modellierende System und seine IT-Security konzentrieren kann. Außerdem bietet es eine Abstraktionsmöglichkeit von konkreten Sicherheitsmechanismen, die erst im weiteren Verlauf der Entwicklung genauer spezifiziert werden. Diese Schutzmaßnahmen gelten für alle Instanzen einer Komponente. So ist ein Bankautomatentyp für einen Angreifer immer genauso schwer zu öffnen, wenn er davor steht.

Das Trust Level wird wie in Listing 4.2 durch das Schlüsselwort `trustlevel` bzw. die Abkürzung `TL` im graphischen Modell definiert, wie in Abbildung 4.1 dargestellt. Der Wert des Trust Level einer Komponente wird dabei relativ zu ihrer Oberkomponente angegeben. Das heißt, wenn eine Komponente über Schutzmaßnahmen verfügt, hat sie ein positives relatives Trust Level und wenn sie beispielsweise Programmcode ausführt, der von einem Angreifer kontrolliert werden kann, ist das relative Trust Level negativ. Wird für eine Komponente kein Trust Level angegeben, gilt für sie das selbe Trust Level wie für ihre Oberkomponente. Die Umgebung des Systems wird grundsätzlich als nicht vertrauenswürdig eingestuft und daher ist ihr relatives Trust Level -1. Zu einer `trustlevel` Definition kann optional noch eine Begründung

für das Trust Level angegeben werden, so wie in Zeile 4 in Listing 4.2 bzw. Abbildung 4.1 dargestellt. Da das Trust Level explizit von einzelnen Schutzmaßnahmen abstrahiert müssen diese Schutzmaßnahmen natürlich korrekt funktionieren und die Komponente tatsächlich schützen, damit die Abstraktion zulässig ist. Die Begründung gibt einem Auditor bei der Überprüfung der Security-Architektur die Möglichkeit zu bewerten, ob das Trust Level gerechtfertigt ist und die vorhandenen Schutzmaßnahmen tatsächlich wirken, oder ob es lediglich dazu dient Fehler, die von automatisierten Analysen erkannt wurden zu vertuschen und nicht zu beseitigen.

Neben dieser strikt hierarchischen Definition des Trust Levels einer Komponente bietet das Schlüsselwort `trustlevelrelation` die Möglichkeit das Verhältnis des Trust Levels zweiter Komponenten anzugeben. In Zeile 5-7 von Listing 4.2 und Abbildung 4.1 sind die Einsatzmöglichkeiten von `trustlevelrelation` dargestellt, die angeben, dass eine Komponente ein höheres Trust Level als eine andere hat oder beide das selbe Trust Level besitzen. Die Definition einer `trustlevelrelation` geschieht immer in einer Komponente, in der die beiden Komponenten auf die sich die `trustlevelrelation` bezieht enthalten sind, sodass der Bezug zu diesen Komponenten klar ist.

Die Angabe des Trust Level und der `trustlevelrelation` müssen widerspruchsfrei sein, damit sie eine Partielle Ordnung $<_{tl}$ definieren, sodass $x <_{tl} y$ bedeutet, dass die Komponente x ein geringeres Trust Level hat als die Komponente y. Um die Analyse von Modellen zu vereinfachen werden jedoch nicht die Relationen gespeichert, sondern jeder Komponente eine Zahl aus $\mathbb{Z}$ als Trust Level zugewiesen, die alle Trust Level Relationen erfüllt. Im Rahmen dieser Berechnung wird auch die Wiederspruchsfreiheit der `trustlevelrelation` geprüft.

Die Abstraktion des Trust Levels beruht darauf, dass Komponenten, Ports und Konnektoren auch in der Implementierung ihre Integrität behalten und Komponenten tatsächlich nur über ihre Schnittstellen kommunizieren können. Daher werden Aussagen, die auf Basis dieser Abstraktion getroffen wurden wertlos, sobald Vulnerabilities in der Implementierung, wie beispielsweise Injections, es einem Angreifer ermöglichen beliebigen Programmcode in einer Komponente auszuführen. Das Trust Level ermöglichen es also Compartmentalization in der Architekturbeschreibung zu definieren, die in der Implementierung eingehalten werden muss.

Es gibt mehrere Ansätze, die das Konzept eines Trust Levels verwenden. In Data Flow Diagrams (DFDs) gibt es das Sprachelement Trust Boundary, welches den Übergang zwischen vertrauenswürdieren und nicht vertrauswürdiegen Komponenten markiert. Eine Trust Boundary verfolgt das selbe Ziel wie das Trust Level in MontiSecArc, allerdings gibt ein Trust Level zusätzlich an, welche Komponente als vertrauenswürdiger betrachtet wird. Diese Information fehlt bei DFDs. Um STRIDE auf Component & Connector-Architekturen anwenden zu können stellen Abi-Antoun und Barnes die Verbindung zwischen DFDs und Component & Connector-Architekturen her [AAB10]. Dabei übersetzen sie Datenflüsse in Konnektoren, Datenspeicher und Prozesse in Komponenten und führen ein TrustLevel für Komponenten ein. Zur Analyse extrahieren sie die Architektur aus der Struktur des zur Laufzeit existierenden Objektgraphen, wodurch es schwierig ist die vom Entwickler intendierten Komponenten im Objektgraphen zu identifizieren.

Trust Level korrespondieren außerdem mit den Security Levels des Bell-LaPadula Modell [BL73], die ein System und dessen Nutzern verschiedene Level zuweist und in einer klassischen

MLS Policy (vergleiche Kapitel 2) nur Informationsflüssen in höhere Security Level zulässt. Eine strikte Anwendung dieser Policy würde jedoch bedeuten, dass ein System stets nur Informationen aufnehmen aber niemals an seine Nutzer abgeben kann.

Security Level wurden mit der Architekturbeschreibungssprachen SADL [MR97] beschrieben und es wurde untersucht wie das Refinement einer solchen Architektur sich auf die Security Level auswirkt. Damit eine MLS Policy sowohl vor als auch nach eines Refinements gültig ist darf es keine zusätzlichen Informationsflüsse zwischen Komponenten einführen [MQRG97]. Direkte Informationsflüsse (Konnektoren) dürfen jedoch durch einfügen einer neuen Komponente in diesen Informationsfluss in indirekte Informationsflüsse verfeinert werden.

Der Multiple Independent Levels of Security and Safety (MILS) Architekturstil [AFHOT06], verwendet ebenfalls den Begriff Security Level. Ziel des MILS Ansatzes ist es ein System in Komponenten (welche in MILS Partitionen genannt werden) zu zerteilen, deren Security-Eigenschaften unabhängig voneinander betrachtet werden können und auf Basis dieser Einzelbetrachtungen eine Aussage über die Security-Eigenschaften des Gesamtsystems treffen zu können. Gelingt diese Komposition, wird es möglich einzelne Komponenten eines MILS Systems auszutauschen und bei einer erneuten Auditierung und Zertifizierung des Gesamtsystems die Teilergebnisse der unveränderten Komponenten wiederzuverwenden. Somit würden die Kosten für eine manuelle Auditierung deutlich sinken.

Um dies zu erreichen werden Komponenten danach unterschieden zu wie viele Security Levels die Daten gehören, die sie verarbeiten:

1. **SLS** Single-Level Secure Komponenten verarbeiten nur Daten aus einem Security Level,
2. **MSLS** Multiple Single-Level Secure Komponenten verarbeiten Daten aus mehren Security Levels, halten diese jedoch zu jeder Zeit getrennt und
3. **MLS** Multi-Level Secure Komponenten verarbeiten Daten aus mehren Security Levels und übertragen auch Daten zwischen diesen gemäß einer Policy, die die Komponente durchsetzt.

Eine MLS Komponente überträgt beispielsweise über eine verschlüsselte Verbindung Daten zu einer anderen MLS Komponente oder filtert Daten durch einen restriktiveren Datentyp auf einer Verbindung zu einer Komponente eines niedrigeren Security Levels.

Auch das Verhalten dieser Eigenschaft bei Refinement einer Komponente durch Subkomponenten wurde untersucht [ZAF08]. Dabei ergeben sich je nach Typ der Oberkomponente und der Verdrahtung der Subkomponenten unterschiedliche Möglichkeiten welchem Typ die Subkomponenten angehören können. Durch diese notwendige Fallunterscheidung ergibt sich keine einfache Kompositionalität für die Eigenschaften SLS, MSLS und MLS.

Die Architekturbeschreibungssprache AADL [FGH06] bietet die Möglichkeit Security Level aus MILS darzustellen und mithilfe der Werkzeuge Ocarina [Hug15] und der Open Source AADL Tool Environment (OSATE) [Sof15] daraus Zugriffskontroll Policies zu generieren, die dem Least Privilege Desing Prinzip folgen bzw. zu überprüfen, ob die definierte Zugriffskontroll Policy diesem Prinzip entsprechen [HWF+10]. Außerdem wurde mit Ocarina eine Implementierung für das MILS Framework POK aus dem AADL Modell generiert, die Partitionen entsprechend der Security Level aus dem AADL Modell anlegt und Verbindungen zwischen diesen Partitionen

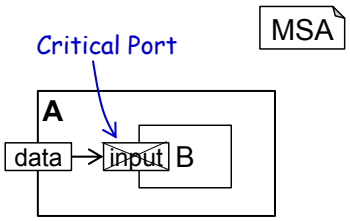


Abbildung 4.3.: Grafische Darstellung eines Critical Port.

mit den definierten verschlüsselten Verbindungen verbindet [HFM08]. Leider scheint diese Werkzeugunterstürzung in aktuellen Versionen nicht mehr vorhanden zu sein und auch POK existiert nicht mehr.

Auch Informationsfluss Policies lassen sich auf Basis von Trust Levels und Konnektoren beschreiben. Wird zusätzlich der Datentyp eines Konektors als Informationsfilter verstanden der von einer Komponente eingehalten wird, lassen sich noch detailliertere applikationsspezifische Informationsfluss Policies definieren [CvdM15].

4.1.2. Critical Port

MSA

```
1 package de.rwth.se.secarc.syntax;
2
3 component A {
4   port in Data data;
5
6   connect data -> b.input;
7
8   component B b {
9     port critical in Data input;
10  }
11 }
```

Listing 4.4: Textuelle Syntax eines Critical Port.

Um eine Security-Architektur auf Weaknesses analysieren zu können ist es wichtig das Ziel eines Angreifers, also die Kernfunktionalität des Systems die es zu schützen gilt, zu kennen. Für eine Analyse ist außerdem wichtig, wie auf diese Kernfunktionalität zugegriffen werden kann. Da Ports den Zugriff auf Funktionalität erlauben, wird in MontiSecArc das Sprachelement `critical port` eingeführt, welches für Ports verwendet wird, die den Zugriff zur Kernfunktionalität des Systems erlauben. Abbildung 4.3 und Zeile 9 in Listing 4.4 zeigen eine Komponente B mit einem solchen `critical port`. In einer guten Security-Architektur sollten `critical ports` nur an Komponenten mit einem höheren Trust Level vorkommen. In einer solchen Archi-

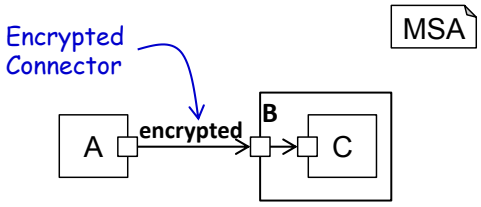


Abbildung 4.5.: Grafische Darstellung des Encrypted Connector.

tektur sind schützenswerte Funktionen nur über `critical ports` erreichbar, welche durch Schutzmaßnahmen geschützt werden.

Vergleichbar mit dem `critical port` ist das in CORAS [LSS10] verwendete Konzept eines Assets. Genauso wie ein `critical port` in MontiSecArc stellt ein Asset ein Angriffsziel eines ebenfalls in CORAS modellierten Angreifers dar.

4.1.3. Encrypted Connector

MSA

```
1 package de.rwth.se.secarc.syntax;
2
3 component A a {
4   port out SecretData;
5 }
6
7 connect encrypted a.secretData -> b.secretData;
8
9 component B b {
10   autoconnect encrypted port;
11   port in SecretData;
12
13   component C {
14     port in SecretData;
15   }
16 }
```

Listing 4.6: Textuelle Syntax von Encrypted Connectoren.

Um die Confidentiality und Integrity von Nachrichten bei der Übertragung über Verbindungen, die ein Angreifer beeinflussen kann, zu gewährleisten, müssen Nachrichten verschlüsselt und signiert werden und der Empfänger muss sie entschlüsseln und die Signatur prüfen. Es gibt verschiedenste Verschlüsselungs- und Signatur-Algorithmen, von denen MontiSecArc abstrahiert. Des Weiteren vernachlässigt MontiSecArc absichtlich die Unterscheidung zwischen signieren und verschlüsseln, da allein die korrekte Verwendung von einzelnen Algorithmen und auch

der richtige Einsatz von Signaturen und Verschlüsselung Security-Wissen voraussetzt, welches gemäß Anforderung A3 nicht notwendig sein soll, um MontiSecArc zu verwenden. Durch das Sprachelement `encrypted connector` welches in Abbildung 4.5 und Listing 4.6 Zeile 7 dargestellt ist stellt MontiSecArc eine vereinfachte Darstellung bereit, die eine Verbindung zwischen zwei Ports modelliert, auf der Confidentiality und Integrity von Nachrichten bei der Übertragung sichergestellt werden und Nachrichten nicht durch einen Angreifer verändert oder gelesen werden können.

Als Erweiterung für das MontiArc Sprachelement `autoconnect` existiert in MontiSecArc `autoconnect encrypted` (Listing 4.6 Zeile 10) welches ausdrückt, dass alle Ports innerhalb der Komponente mit diesem Statement, bei denen die Typen übereinstimmen mit einem `encrypted connector` verbunden sind. Obgleich die vollständige Verschlüsselung jeglicher Kommunikation in Zeiten von globaler Überwachung durch Geheimdienste eine gute Idee ist, bedeutet jede verschlüsselte Kommunikation Wartungskosten für die Kommunikationsendpunkte, da die verwendeten Schlüssel und Zertifikate regelmäßig erneuert werden müssen. Außerdem können Confidentiality und Integrity auch durch physische Schutzmaßnahmen gewährleistet werden, die verhindern, dass ein Angreifer die Verbindung beeinflussen kann. Bei den meisten Systemen wird es ausreichen, wenn CPU und Arbeitsspeicher unverschlüsselt kommunizieren, aber wenn der Angreifer Zugriff auf diese Kommunikation erlangen kann, muss auch diese Übertragung geschützt werden. Es gibt also keine optimale Lösung, die für alle Systeme gilt, sondern Entwickler müssen entscheiden ob die Übertragung von Nachrichten abgesichert werden soll.

Auch für xADL [DvdHT02] existiert eine Erweiterung, die Security-Eigenschaften in diese Component & Connector-Architekturbeschreibungssprache einführt. Dabei werden jedoch nur verschlüsselte Konnektoren [RTDR05] und Zugriffskontrolle eingeführt [RT06]. Ein aktuelles Survey [FKK+14, S. 14] kommt zu dem Schluss, dass außer diesem Ansatz und DFDs im Bereich Threat Modeling keine ausreichende Security-Architekturbeschreibungssprache existiert, welche alle Aspekte in einer Component & Connector-Architektur modelliert.

4.1.4. Authorization

Wie bereits in Kapitel 2 beschrieben wird, ist Authorization (Definition 2.1.2) eine wichtige Schutzmaßnahme, um Angreifern den Zugriff auf Funktionen und Daten eines Systems zu verweigern, während legitimen Nutzern der Zugriff gewährt wird. Um Authorization zu realisieren existieren wie bei Verschlüsselung und Signatur mehrere Verfahren, wie z.B. Access Control Lists und Role Based Access Control. Jedes dieser Verfahren definiert welche Access Policy für welchen Nutzer des Systems gilt. Entsprechend der Access Policy gewährt ein System einen Zugriff bzw. verweigert ihn. MontiSecArc abstrahiert von diesen unterschiedlichen Verfahren und repräsentiert Authorization durch das Sprachelement `access` gefolgt von einem Access Policy Namen. Der Access Policy Name referenziert die entsprechende Access Policy des verwendeten Authorization Verfahrens, sodass das Sprachelement `access` festlegt, dass der Zugriff nur für Nutzer erlaubt ist, die das Authorization Verfahren in der entsprechenden Access Policy erlaubt. Die Verwaltung von Benutzern und die Definition von Access Policies gehört somit zu den Aufgaben des Authorization Verfahrens und wird daher nicht in MontiSecArc betrachtet. Van den Berghe et al. [vdBSYJ15] zeigen das es eine Vielzahl von modellgetriebenen Ansätzen gibt,

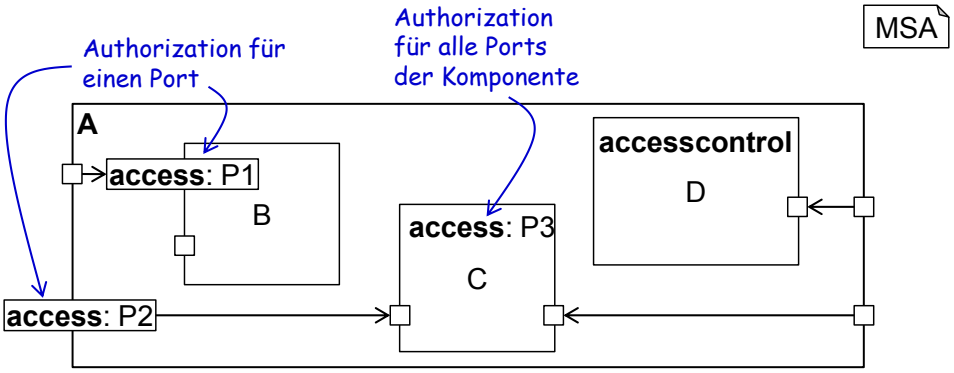


Abbildung 4.7.: Grafische Darstellung der Zugriffskontrolle.

die hierzu verwendet werden können. Wie in Abbildung 4.7 und Listing 4.8 dargestellt wird, kann `access` entweder für einzelne Ports verwendet werden (Zeile 14) und dort den Zugriff für individuelle Ports beschränken oder innerhalb einer Komponente (Zeile 18), wodurch die Access Policy für alle eingehenden Ports der Komponente gilt.

Wie in diesem Beispiel dargestellt, kann `access` sowohl in der Superkomponente A als auch in den Subkomponenten B und C definiert werden. Dabei findet die Authorization im Falle von B nur in B und nicht in A statt, wohingegen für den Zugriff auf C sowohl die Access Policy P2 in A als auch die Policy P3 in C geprüft wird. Durch Verfeinerung in einer Subkomponente kann die Zugriffskontrolle einer Oberkomponente also nur eingeschränkt werden. Um die Kompatibilität mit MontiArc zu gewährleisten dient `access` lediglich dazu den Zugriff einzuschränken, sodass ohne Angabe von `access` der Zugriff ohne Authorization möglich ist.

Besonders zu Beginn des Architekturentwurfs sind Komponenten und Ports an denen Authorization durchgeführt werden soll unter Umständen noch gar nicht definiert, genauso wenig wie genaue Access Policies die dort gelten sollen. Wenn in dieser frühen Phase des Entwurfs allerdings bereits feststeht, dass innerhalb einer bestimmten Komponente Authorization durchgeführt werden muss kann dies mittels des Sprachelements `accesscontrol on` definiert werden. Dadurch legt `accesscontrol on` fest, dass Authorization durchgeführt werden muss, lässt aber offen welche Access Policies gelten sollen und welche Ports diese betreffen. Präzise formuliert drückt `accesscontrol on` in einer Komponente A aus, dass A oder eine ihrer Subkomponenten `access` für alle eingehenden Ports von A definieren müssen. Wird `access` an einem Port einer Subkomponente definiert und nicht direkt in A, so ergibt sich daraus direkt die Policy, die auch am eingehenden Port von A gelten muss.

Wie Zugriffskontrolle für eine Component & Connector-Architektur spezifiziert und verfeinert werden kann beschreiben Deng et al. [DWTB03]. Die Zugriffskontrollaspekte werden dabei auf Petrinetze und Temporale Logik abgebildet, allerdings wird keine ADL angegeben.

```
1 package de.rwth.se.secarc.syntax;                                    MSA
2
3 component A {
4   port in DataB1,
5        in DataC1,
6        in DataC2,
7        in DataD;
8
9   access dataC1 (P2); //Am Port dataC1 gilt die Access Policy P2
10
11  component B {
12    port in DataB1 datab1,
13         in DataB2;
14    access dataB1 (P1); //Am Port dataB1 gilt die Access Policy P1
15  }
16
17  component C {
18    access P3; //An allen Ports von C gilt die Access Policy P3
19    port in DataC1,
20         in DataC2;
21  }
22
23  component D {
24    accesscontrol on; //Zugriffskontrolle muss vorhanden sein,
25                      //genaue Access Policy wird offen gelassen
26    port in DataD;
27  }
28
29  connect dataB1 -> B.dataB1;
30  connect dataC1 -> C.dataC1;
31  connect dataC2 -> C.dataC2;
32  connect dataD  -> D.dataD;
33 }
```

Listing 4.8: Textuelle Syntax der Zugriffskontrolle.

4.1.5. Authentication

Die Identität eines Systemnutzers (egal ob Mensch oder Maschine), die für die Authorization verwendet wird ist grundsätzlich nur innerhalb einer Komponente gültig, da nur in einer Komponente durch Schutzmaßnahmen verhindert werden kann, dass ein Angreifer eine falsche Identität vortäuscht. Damit ein Nutzer aus einer anderen Komponente A auf einen mit `access` geschützten Port von Komponente B zugreifen kann muss A in einem Authentication Protokoll B beweisen, dass der Nutzer ist wer er behauptet zu sein. In diesem Authentication Protokoll wird A als Prover und B als Verifyer bezeichnet. In MontiSecArc wird durch einen `identity` Link explizit angegeben zwischen welchen Komponenten eine Authentication stattfindet und somit die Identitäten der Nutzer dieser Komponenten verbunden werden. Der `identity` Link, wie in

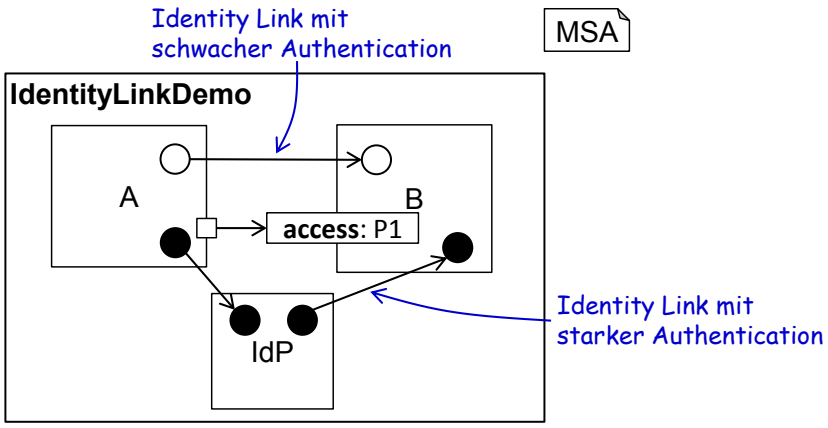


Abbildung 4.9.: Grafische Darstellung von `identity` Links.

```
1  package de.rwth.se.secarc.syntax;
2
3  component IdentityLinkDemo {
4
5    component A a {
6      port out Data;
7    }
8
9    component B b {
10     port in Data;
11     access data (P1);
12   }
13
14   component IdP idp {
15   }
16
17   identity weak a -> b;
18   identity strong a -> idp;
19   identity strong idp -> b;
20   connect a.data -> b.data;
21
22 }
```

MSA

Listing 4.10: Textuelle Syntax von `identity` Links.

Abbildung 4.9 und Zeile 17-19 von Listing 4.10 dargestellt, ist vom Prover zum Verifyer gerichtet, da nur in diese Richtung die Identitäten nachgewiesen werden. Somit wird sichergestellt, dass nur Nutzer, die in A oder einer Subkomponete von A die Policy P1 erfüllen auch in B oder einer Subkomponente von B Zugriff auf Ports erhalten, denen mit `access` diese Policy zugewiesen wurde.

Kryptographisch abgesicherten Verfahren, die mehrere Faktoren wie z.B. das Wissen über

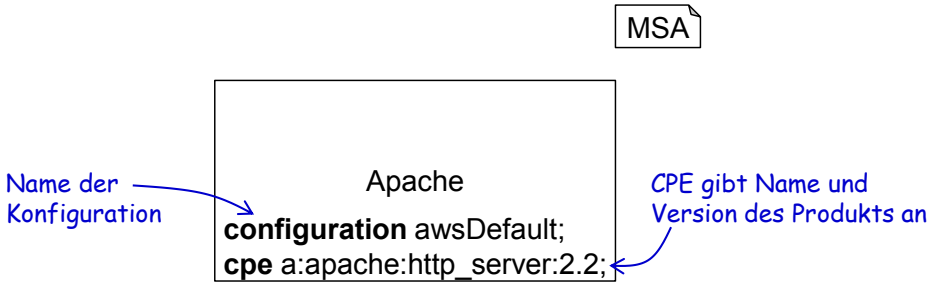


Abbildung 4.11.: Grafische Darstellung von Drittanbieterkomponenten.

eine PIN und den Besitz einer SmartCard oder den Besitz eines Ausweisdokuments zur Authentisierung verwenden bieten einen besseren Schutz als Verfahren, die ein einfaches Passwort unverschlüsselt übertragen. Um diesen qualitativen Unterschied in MontiSecArc zu repräsentieren wird zwischen `strong identity` Link, welcher für starke Mehrfaktor Authentication steht, und `weak identity` Link, welcher für einfache Authentication beispielsweise mit einem Passwort steht, unterschieden. In Abbildung 4.9 und Listing 4.10 in Zeile 17 wird ein `weak identity` Link zwischen Komponente A und B definiert und in Zeile 18 und 19 zwei `strong identity` Links zwischen Komponente A und dem IdP[1], sowie zwischen dem IdP und B. Nicht alle Authentication Verfahren lassen sich klar in eine dieser Kategorien einteilen. Beispielsweise kann Passwort Authentication von einem Rechner aus, der bereits zuvor zur erfolgreichen Authentication verwendet wurde je nach Einschätzung der am Entwurf beteiligten Parteien als `weak` oder `strong` eingestuft werden.

Yoder und Barcalow [YB97] beschreiben das Check Point Security Design Pattern, bei dem die Authorization an einem zentralen Punkt der Applikation definiert wird. Dabei reicht zu Beginn der Implementierung eine Methode aus, die immer den Zugriff erlaubt. Wichtig ist, dass zu einem späteren Zeitpunkt die Authorization genauer spezifiziert werden kann und diese dann in der gesamten Applikation angewendet wird. Genau diese Eigenschaften erfüllt auch `accesscontrol on` zusammen mit `access` in MontiSecArc, allerdings ist hier die Security-Architektur die zentrale Stelle um die Authorization zu definieren.

4.1.6. Drittanbieterkomponenten

Durch die Kapselung von Funktionen in Komponenten wird die Wiederverwendung einzelner Komponenten verbessert, sodass jedes System Drittanbieterkomponenten wie Webserver, Bibliotheken oder ganze Betriebssysteme wiederverwendet. Für diese Komponenten werden in der NVD (siehe Kapitel 2.1.3) bekannte Vulnerabilities gespeichert. In dieser Datenbank werden die Komponenten als Produkte bezeichnet und ein Produkt wird durch eine CPE identifiziert. Da täglich neue Vulnerabilities gemeldet werden, muss insbesondere wenn das System im Einsatz ist regelmäßig überprüft werden, ob neue Vulnerabilities in den wiederverwendeten Komponenten durch einen Angreifer ausgenutzt werden können. Um diesen Aspekt der technischen Architektur

[1] Die Komponente IdP steht hier für einen Identity Provider, welcher als vertrauenswürdige Instanz die Authentication von Nutzern im System regelt [PM03].

```
1 package de.rwth.se.secarc.syntax;                    MSA
2
3 component Apache {
4   cpe "a:apache:http_server:2.2";
5   configuration awsDefault;
6 }
```

Listing 4.12: Textuelle Syntax von Drittanbieterkomponenten.

ebenfalls abbilden zu können, bietet MontiSecArc das Sprachelement `cpe` an, welches in Abbildung 4.11 und Zeile 4 in Listing 4.12 verwendet wird. Dieses dient dazu Komponenten von Drittanbietern explizit zu kennzeichnen und über die CPE des Produkts eindeutig zu identifizieren. Über die CPE ist es somit möglich bekannte Vulnerabilities der verwendeten Komponenten aus der NVD abzufragen und diese entweder durch eine Aktualisierung der Komponente zu beheben oder durch zusätzliche Gegenmaßnahmen in der Architektur die Ausnutzung zu verhindern.

Eine Aktualisierung, bei der nicht nur Vulnerabilities behoben werden, sondern eine Komponente auch neue Funktionalität erhält, kann nicht nur neue, bislang unbekannte Weaknesses mit sich bringen, sondern macht die Komponente unter Umständen auch inkompatibel zu ihrer früheren Version. Die Ausnutzung von Vulnerabilities durch Angreifer lässt sich auch dadurch verhindern, dass die verwundbare Funktionen deaktiviert wird, wenn diese für die Funktion des Gesamtsystems nicht benötigt wird. Daher bietet sich statt einer Aktualisierung eine Anpassung der Konfiguration einer verwundbaren Komponente an. Für die Security-Architektur bedeutet dies, dass die Konfiguration einer Komponente, welche in MontiSecArc mit dem Sprachelement `configuration` modelliert wird, darüber entscheidet ob die Architektur gegen bekannte Vulnerability geschützt ist. Eine Konfiguration wird wie im Beispiel in Zeile 5 durch einen Namen in der Architektur repräsentiert. Auditoren können anhand dieser eindeutig benannten Konfiguration der externen Komponente überprüfen, ob diese die Ausnutzung aller bekannten Vulnerabilities verhindert.

Ein ähnliches Ziel verfolgt MulVal [OGA05], welches die Informationen aus der NVD zusammen mit einem Modell des Netzwerks und der Konfiguration der Hosts in diesem Netzwerk sowie den Zugriffsrechten von Nutzern auf den Hosts nutzt, um Aussagen über die Ausnutzbarkeit von bekannten Vulnerabilities in einem System durch einen Angreifer zu treffen. MulVal verwendet jedoch keine Modellierungssprache, sondern Prolog um die Fakten zu repräsentieren und Aussagen abzuleiten.

4.1.7. Physische Security

Im Vordergrund dieses Kapitels standen bisher Sprachelemente von MontiSecArc, die die Security-Architektur von verteilten IT Systemen beschreiben und von Maßnahmen der physischen Security durch das Trust Level abstrahieren. In Cyber Physical Systems müssen jedoch physische Aspekte von IT Systemen mit in die Security-Architektur einbezogen werden, da Angreifer auf Teile des Systems unmittelbaren Zugriff haben und diese daher mit Hardware Security-Schutzmaßnahmen geschützt werden müssen. Basierend auf existierenden Klassifika-

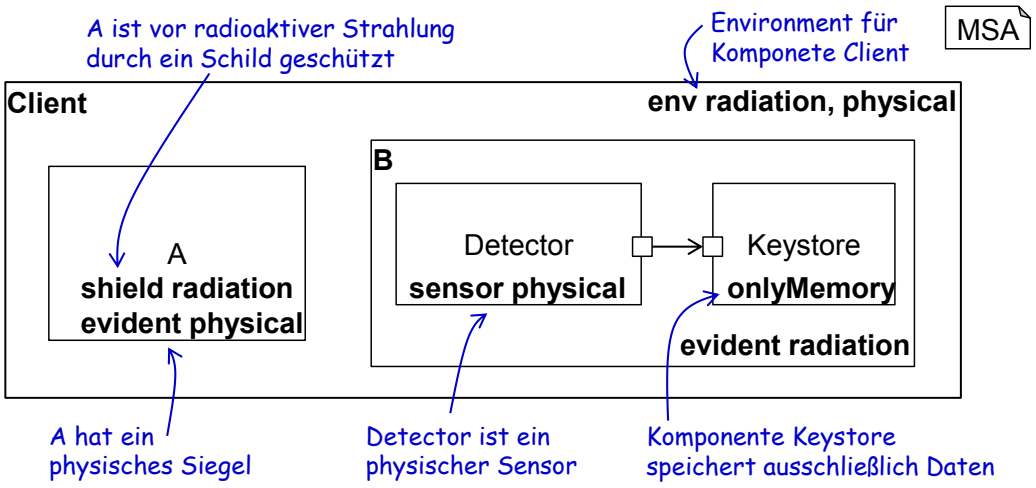


Abbildung 4.13.: Grafische Darstellung von physikalischen Kanälen und Schutzmaßnahmen.

tionen für physische Angriffe und Schutzmaßnahmen auf IT Systeme [Wei00, AK97, Sko05] bietet MontiSecArc mehrere Sprachelemente, mit denen Physische Security-Eigenschaften der Architektur definiert werden können.

Environment

Abseits der vorgesehenen Kommunikationskanäle existieren eine Vielzahl von Kanälen, über die ein Angreifer auf ein System zugreifen kann und Informationen erlangt. Angefangen von physischem Zugang, über das Abhören von Kommunikationsverbindungen bis hin zur Auswertung von Effekten, die ein System während der Berechnung aussendet, den sogenannten Side-Channels. MontiSecArc bietet mit dem Sprachelement `env` (Kurzform für Environment) einen Mechanismus, um zu definieren welche dieser Kanäle in der Security-Architektur berücksichtigt werden sollen. Mit `env` werden in einer Komponente eine bestimmte Menge von Kanälen definiert, wie z.B. physical und radiation in Abbildung 4.13 und Zeile 4 von Listing 4.14, durch die Angriffe auf die Komponente und ihre Subkomponenten als grundsätzlich möglich erachtet werden. Es gibt eine Vielzahl von bekannten Angriffen, die verschiedene physikalische Effekte als Zugangskanal zu einem System ausnutzen. In MontiSecArc sind daher die folgenden Kanäle definiert, die in bekannten Angriffen verwendet werden, um Zugriff auf eine Komponente zu erhalten:

- `physical`: Physischer Zugang, bei dem durch Krafteinwirkung, auch mit Hilfe von Werkzeugen, auf das Innenleben einer Komponente zugegriffen wird.
- `chemical`: Durch chemische Reaktionen mit der Komponente wird die Hülle entfernt und das Innenleben freigelegt.
- `electromagnetic`: Elektromagnetische Strahlung, die in die Komponente eindringt kann ihr Verhalten ändern.

```
1 package de.rwth.se.secarc.syntax;                                    MSA
2
3 component Client {
4   env radiation, physical;
5
6   component A {
7     shield radiation;
8     evident physical;
9   }
10
11  component B {
12    evident radiation;
13
14    component Detector detector {
15      sensor physical;
16      port out Alarm;
17    }
18
19    connect detector.alarm -> keystore.alarm;
20
21    component Keystore keystore {
22      onlyMemory;
23      port in Alarm;
24    }
25  }
26 }
```

Listing 4.14: Textuelle Syntax von physischen Kanälen und Schutzmaßnahmen.

- `radiation`: Radioaktive Strahlung, die in die Komponente eindringt oder sie durchdringt kann sowohl das Verhalten der Komponente ändern, als auch Rückschlüsse auf das Innenleben zulassen.
- `temperature`: Gezielte Temperaturänderung der Komponente beeinflusst die unter Umständen durchgeführten Berechnungen.
- `voltage`: Die Veränderung der Betriebsspannung kann Berechnungen und auch das Verhalten von Speichern beeinflussen.
- `light`: Ähnlich wie radioaktive Strahlung durchdringt Licht manche Materialien, sodass Rückschlüsse auf das Innenleben einer Komponente möglich werden.

Da es jedoch auch möglich ist, dass eine Komponente über Seitenkanäle Informationen preisgibt, ohne dass sie von einem Angreifer geöffnet werden muss wird `side-channel` als zusätzlicher Kanal definiert. Auf Ebene der Security-Architektur verhalten sich diese im Detail unterschiedlichen Kanäle sehr ähnlich, sodass diese gemeinsame Abstraktion möglich ist.

Durch `env` wird für eine Komponente eine Menge von Kanälen definiert, über die der Zugriff auf diese Komponente möglich ist. Für all diese Kanäle müssen Schutzmaßnahmen in der Komponente eingebaut werden, damit kein Kanal für einen Angriff ausgenutzt werden kann.

Das Sprachkonzept von `env` ähnelt dem von `accesscontrol on`, indem es von den konkreten Angriffen abstrahiert und durch seinen Einsatz der Entwickler dazu angehalten wird Schutzmaßnahmen in die Architektur zu integrieren. Genauso wie für `accesscontrol on` gibt es für `env` Analysen, die in Abschnitt 5.2.3 beschrieben werden und erkennen, wenn für einen der mit `env` angegebenen Kanäle keine Schutzmaßnahme vorhanden ist.

Physische Schutzmaßnahmen

Durch eine Kategorisierung der Schutzmaßnahmen gegen physische Angriffe ergaben sich die im Folgenden beschriebenen Sprachelemente `shield`, `evident`, und `sensor` für MontiSecArc. Grundsätzlich lässt sich jede dieser Schutzmaßnahmen für jeden der zuvor beschriebenen Kanäle einsetzen. Hinter dem Sprachelement, welches die Schutzmaßnahme in MontiSecArc repräsentiert wird der Kanal angegeben, für den diese wirkt. Wirkt eine Gegenmaßnahme gegen Angriffe auf mehreren Kanälen die durch `env` als zu berücksichtigen markiert wurden, so werden diese zusammen angegeben.

- `shield` (Schutzschild) verhindert, dass ein Angriff auf den angegebenen Kanälen die Komponentengrenze überqueren kann. Somit ist ein Angreifer nicht in der Lage die Komponenten und Konnektoren innerhalb der Komponente anzugreifen. Klassische Beispiele für ein solches `shield` sind Schutzhüllen, die sich nicht öffnen lassen ohne den darin enthaltenen Chip zu zerstören oder Abschirmungen gegen radioaktive Strahlung. In Abbildung 4.13 und in Zeile 7 von Listing 4.14 wird ein `shield` für `radiation` definiert, welches die Komponente A schützt.

- `evident` (Hinweis) verhindert nicht den Angriff auf eine Komponente, sondern bemerkt diesen und hinterlässt einen Hinweis der vom Angreifer nicht entfernt werden kann, sodass bei einer Überprüfung des Systems schnell offensichtlich ist, dass es kompromittiert wurde. Diese Gegenmaßnahme setzt nicht auf Prävention der Kompromittierung eines Systems, sonder darauf, dass ein Systemnutzer vor der Nutzung des Systems die Kompromittierung erkennt, das System nicht verwendet und somit der Angriff nicht erfolgreich ist. Wenn `evident` zusätzlich noch Information, die zur Identifikation des Angreifers dienen, speichert können die organisatorischen und gesellschaftlichen Sanktion die ein Angreifer im Falle eines Angriffs auf das System erfährt, als Abschreckung dienen. In Abbildung 4.13 sowie in Zeile 8 und 12 von Listing 4.14 werden `evident` für `physical` bzw. `radiation` definiert, sodass die Komponente A eine Art Siegel besitzt, welches ein Angreifer aufbrechen muss, um an das Innere zu gelangen und Komponente B eine Substanz verwendet, die sich bei radioaktiver Bestrahlung sichtbar verändert.

- `sensor` erkennt genauso wie `evident` einen Angriff, bleibt dann allerdings nicht passiv, sondern sendet eine Nachricht, die von den Empfängern genutzt wird um Gegenmaßnahmen auszulösen. Üblicherweise ist das Ziel dieser Maßnahmen die für einen Angreifer wertvollen Informationen zu löschen, bevor dieser lesenden Zugriff erlangt. Durch das

Sprachelement `sensor` innerhalb einer Komponente wird diese zum Sensor deklariert, der über seine ausgehenden Ports die Information über einen gerade stattfindenden Angriff sendet. In Abbildung 4.13 und in Zeile 15 von Listing 4.14 wird die Komponente Detector zum Sensor für den Kanal `physical` bestimmt, sodass bei einem physischen Angriff auf B eine Alarm Nachricht an den Keystore gesendet wird.

Diese Schutzmaßnahmen unterscheiden sich in Kosten ihrer Realisierung und Ressourcen zur Laufzeit, die für ihren Betrieb notwendig sind. `shield` und `evident` sind passive Techniken, die auch ohne Ressourcen wie Energie und Kommunikationskanäle ihre Schutzwirkung besitzen. Der `sensor` hingegen benötigt zumindest eine aktive Gegenmaßnahme, die über einen Konnektor aktiviert wird. Betrachtet man den Grad zu dem die Gegenmaßnahmen einen Angriff abwehren, so stellt man fest, dass `shield` den Angriff vollständig unmöglich macht, `sensor` versucht ihn zu verhindern nachdem er erkannt wurde und `evident` verhindert den Angriff überhaupt nicht, sondern zeigt nur die Kompromittierung des Systems an.

Somit müssen je nach Funktionalität eines Systems entsprechende Schutzmaßnahmen ausgewählt werden, die wirksamen Schutz bieten und kostengünstig sind. Da bestimmte Angriffe, wie beispielsweise Änderung der Betriebsspannung, nur bei der Durchführung von Berechnungen in CPUs und andere, wie beispielsweise das Festbrennen und Extragieren von Informationen, nur auf Speicher angewandt werden können wird durch die Schlüsselwörter `onlyCPU` und `onlyMemory` eine Komponente gekennzeichnet, die ausschließlich Berechnungen durchführt bzw. nur Daten speichert. Wird keines dieser Schlüsselwörter angegeben, so wird bei der Analyse davon ausgegangen, dass eine Komponente sowohl eine CPU als auch Speicher besitzt. In Abbildung 4.13 und in Zeile 22 von Listing 4.14 wird für die Komponente Keystore definiert, dass diese ausschließlich zur Datenspeicherung verwendet wird und keine Berechnungen durchführt.

4.2. Weitere Verwandte Arbeiten

Existierende Modellierungssprachen die Security-Eigenschaften modellieren und im Softwareentwurf genutzt werden können sind nicht auf die hier verwendeten Component & Connector-Architekturbeschreibungssprachen beschränkt. In einem Survey von van den Berghe et al. [vdBSYJ15] wurden 28 dieser Sprachen nach verschiedenen Kriterien klassifiziert. Betrachtet man die Ausdrucksstärke bzgl. Security-Eigenschaften, so fällt auf, dass 20 der 28 untersuchten Sprachen Zugriffskontrolle unterstützen und 15 von diesen keine anderen Security-Eigenschaften darstellen können. Wie die Vielzahl an unterschiedlichen Vulnerabilities, die in den CWE gelistet werden zeigt, reicht es nicht aus Zugriffskontrolle korrekt zu implementieren um ein sicheres System zu entwickeln, daher unterstützt MontiSecArc weitere Security-Eigenschaften, die über die Zugriffskontrolle hinausgehen.

Im Folgenden werden solche Ansätze, die wie in Anforderung A1 gefordert, ebenfalls einen Agilen MDD Ansatz mit Codegenerierung unterstützen und neben der Zugriffskontrolle noch weitere Aspekte berücksichtigen dargestellt.

UMLsec [Jür02, Jür05a] ist ein UML Profil, welches die in der UML zur Spracherweiterung vorgesehenen Stereotypen nutzt, um Security-Eigenschaften in verschiedene Diagramme der UML einzufügen. Dabei werden abstrakte Konzepte wie `fair exchange, secure`

`link`, `data security`, `secure dependency`, die Security-Terminologie `secrecy`, `integrity`, abstrakte Zugriffskontrolle `guarded access`, `guarded`, konkrete Zugriffskontrollmodelle `RBAC`, konkrete Informationsfluss Policies `no down-flow`, `no up-flow`, sowie konkrete Arten von Verbindungen wie `LAN` und `Internet` verwendet. Durch ein Angreifermodell wird festgelegt, welche der Aktionen `delete`, `read`, `insert` ein Angreifer in einem durch einen Stereotyp markierten Bereich oder Verbindung ausführen kann. Auf Basis dieser Spezifikation lässt sich berechnen, ob der modellierte Angreifer die Security-Eigenschaften eines Systems kompromittieren kann [Jür05b]. Dazu wird die Spezifikation in First-Order Logic überführt und mittels eines Theorem-Prover überprüft. Dieses Vorgehen wird verwendet, um in Sequenzdiagrammen und Statecharts modelliertes Verhalten zur Analyse von Protokollen wie TLS zu verwenden [Jür05b, BJN07, JS07]. Außerdem lassen sich die Verbindungen in UML Deployment Diagrammen daraufhin überprüfen, ob die Confidentiality Anforderungen für verschiedene Angreifermodelle erfüllt sind [JSB08].

In der UMLsec Sprache werden viele Instanzen von eigentlich allgemeineren Konzepten verwendet. Zum Beispiel sind `LAN` und `Internet` zwei konkrete Ausprägungen eines Netzwerks bzw. einer Verbindung. Somit muss die Sprache, das Angreifermodell und die werkzeugunterstützte Analyse erweitert werden, sobald diese Instanzen nicht ausreichen um eine System zu modellieren. Dieser Fall tritt auf, sobald ein System mit verschiedenen Netzwerksegmenten modelliert werden soll. Entwickler die UMLsec nutzen, müssen daher gleichzeitig auch UMLsec und die verwendeten Werkzeug für ihre Zwecke erweitern.

Extended Data Flow Diagrams [BSK16] erweitern die in 2.2 erläuterten Data Flow Diagrams (DFDs) und führen Trust Areas, ein Typsystem und Annotationen zur ad-hoc Spracherweiterung ein. Durch Trust Level, das MontiArc Typsystem und Stereotypen bietet MontiSecArc die gleichen Konzepte. Die Möglichkeit Schutzmaßnahmen in MontiSecArc zu definieren erlauben es jedoch eine detailliertere Analyse durchzuführen, die nicht alle möglichen Angriffe auflistet, sondern nur solche die nicht durch die Schutzfunktionen verhindert werden.

Hoisl et al. [HSS14] verwenden UML Aktivitätsdiagramme um den Daten und Kontrollfluss in Businessprozessen zu beschreiben, die mit WS-BPEL [OAS07] zur Ausführung gebracht werden. In diesen Prozessen soll sichergestellt werden, dass Informationen, die einmal einen Confidentiality- oder Integrity-Schutz erhalten haben, also schützenswert sind, diesen im restlichen Prozess beibehalten. Um diese Eigenschaft zu prüfen, werden Regeln definiert nach denen die Confidentiality und Integrity eines gesamten Aktivitätsdiagramms aus den Confidentiality und Integrity Eigenschaften der einzelnen Aktivitäten und Datenspeicher bestimmt werden. Durch die Integration in das UML4SOA Profil [MKSK10] wird eine automatische Verarbeitung der Aktivitätsdiagramme zu WS-Policy [OAS12] Dokumenten möglich, die definieren, welche konkreten Algorithmen zum Schutz von Confidentiality und Integrity angewandt werden sollen.

Analog zu MDA beschreiben Memon et al. [MMD+14] einen Ansatz, der Security-Pattern auf verschiedenen Abstraktionsstufen einsetzt und sie somit verfeinert. Die konkretisierten Pattern werden dann automatisch in WS-Policy Dokumente überführt, die beschreiben welche Schutzmaßnahmen eingesetzt werden sollen, um ein auf Webservices basierendes System zu schützen.

Nakamura et al. [NTIO05, SNO06] erzeugen ebenfalls WS-Policy Dokumente aus Modellen, um es Entwicklern ohne Security-Knowhow zu ermöglichen komplexe Authentication-Protokolle

in ihren Applikationen zu verwenden. Dazu definieren sie für Klassen im Datenmodell einer Applikation die Stereotypen `Confidentiality, Integrity, Authentication` und `Non-repudiation`. Mit Hilfe von UML Deployment Diagrammen, die die Component & Connector-Architektur des Zielsystems darstellen und einem Security Infrastructure Model, welches die in der Implementierung vorhandenen WS-Policy Features definiert werden diese Stereotypen in WS-Policy Dokumente übersetzt.

MetaH [BEJV96] setzt Security zur Laufzeit eines Betriebssystems um, indem die laufenden Prozesse nur über die in MetaH definierten Schnittstellen miteinander kommunizieren dürfen.

Neben diesen konstruktiven Security-Ansätzen zur Modellierung von Software Security gibt es mehrere Modellierungssprachen, die von Security-Experten zur Beschreibung von Angriffsszenarien genutzt werden. Diese eigenen sich jedoch nicht für Entwickler (Anforderung A3), da Angreifer, Bedrohungen und Eintrittswahrscheinlichkeiten von Angriffen modelliert werden, welche in der konstruktiven Entwicklersicht nicht existieren. Attack Trees [Sch99], waren ein erster Ansatz diese Angriffe systematisch zu modellieren und daraus abzuleiten wie wahrscheinlich bzw. schwer ein Angriff ist. Diese Art der Security Risk Analysis wird unter Anderem im CORAS Ansatz [LSS10, SS13] weiter verfeinert. Dabei werden mehrere Modellierungssprachen eingesetzt, um die Sicht eines Security-Experten auf ein System systematisch darzustellen und aus dieser Darstellung Risks für das System bzw. Angriffswahrscheinlichkeiten abzuleiten. Die hierzu verwendeten Modellierungselemente sind Assets also die zu schützenden Werten, Angreifer, Vulnerabilities, die Wirkung von Vulnerabilities auf Assets, Nutzer, Risks und Gegenmaßnahmen. Um das Risk für ein System zu errechnen, werden die Eintrittswahrscheinlichkeit der unterschiedlichen Vulnerabilities wie bei Attack Trees geschätzt und ihre Auswirkungen auf die Assets betrachtet. Diese Eintrittswahrscheinlichkeiten lassen sich nur schwer schätzen, was eine grundsätzliche Schwäche des Ansatzes ist.

Von der Sichtweise her ähnlich wie Attack Trees modellieren Defense Graphs [SL15] die Schutzmaßnahmen, welche eine Anfrage während der Verarbeitung durch ein System durchläuft. Dabei werden als wesentliche Schutzmaßnahmen ein Policy Enforcer und ein Policy Selector identifiziert und die Auswirkungen der Komposition von redundanten und unabhängigen Schutzmaßnahmen betrachtet. Policy Enforcer entscheiden anhand einer gegebenen Policy, ob eine Nachricht die sie erhalten weitergeleitet werden soll. Zugriffskontrolle und Firewalls sind demnach Policy Enforcer. Policy Selector entscheiden anhand des Typs einer erhaltenden Nachricht bzw. einer darin enthaltenen Information wie diese weiterverarbeitet werden soll. Ein Router, der anhand der Adressen in Netzwerkpaketen entscheidet wie diese weitergeleitet werden sollen ist demnach ein Policy Selector. Defense Graphs beschreiben somit wesentliche Aspekte der Security-Architektur, allerdings fehlen bislang Modellelemente um beispielsweise Verschlüsselung abzubilden.

Secure Tropos [MG07], ein Ansatz aus dem Bereich der Anforderungserhebung, benennt auch die Konzepte `Security Feature`, `Protection Objective`, `Security Mechanism` und `Threat`, welche in der Architektur angewandt werden können.

Für die Cyber Security Modeling Language (CySeMol) [SEH13], welche ebenfalls verwendet wird um die Wahrscheinlichkeit von Angriffen zu berechnen, wird ein umfassendes Metamodell angegeben, welches Schutzmaßnahmen und Systemkomponenten beschreibt. Dabei werden

physikalische und Netzwerk Zonen, Datenflüsse und das verwendete Netzwerkprotokoll sowie konkrete Netzwerkschutzmaßnahmen wie Firewall, Intrusion Detection System und Deep Packet Inspection modelliert. Für die Software im System werden Zugriffskontrolle, das Betriebssystem und die bekannten Vulnerabilities der eingesetzten Applikationen modelliert. CySeMol richtet sich eindeutig an Netzwerk Security-Experten und verwendet das in dieser Nutzergruppe verbreitete Vokabular. Softwareprodukte werden nur als ganzes modelliert und auch nicht ihre Eigenschaften, die zu Vulnerabilities führen.

Ein anderer Ansatz, der im Grunde die gleiche Sichtweise wie Attack Trees darstellt sind Misuse und Abuse Cases [Ale03, HMA04]. Diese basieren auf UML Use Cases und erweitern diese um Akteure, die versuchen das System zu ihrem Vorteil zu missbrauchen. Somit werden Szenarien beschrieben, die nicht eintreten sollen.

Wie bereits in Kapitel 2.2 erläutert wurde verwenden der Microsoft SDL DFDs und die Architectural Risk Analysis von Cygitel informelle graphische Component & Connector Modelle zur Darstellung der Struktur eines Systems, um darauf bekannte Angriffstechniken durch Security-Experten anzuwenden [Del15a, Del15b]. Die Ergebnisse dieser Vorgehensweise sind sehr vielversprechend, allerdings müsste bei einem agilen Vorgehen in jedem Sprint eine Security-Architekturanalyse durchgeführt werden, was bei einer manuellen Analyse einen erheblichen Zeitaufwand für Security-Experten und somit Kosten für ein Projekt bedeutet. Um also die Anforderungen A2 umzusetzen wird hier der Ansatz der agilen modellgetriebenen Entwicklung verwendet, um die Effizienz des Entwicklungsprozesses durch Werkzeugunterstützung zu verbessern und somit auch in Projekten ohne Security-Experten Architectural Risk Analysis durchführen zu können.

Zusammenfassend lässt sich feststellen, dass sich diese Ansätze an Security-Experten richten, die ein System analysieren und bewerten. Diese Zielgruppe nutzt die Ansätze, um ihre intuitive, destruktive Vorgehensweise zu strukturieren. Da Entwickler jedoch eine konstruktive Sicht auf das System haben und nur schwer die Perspektive eines Angreifers einnehmen können, um Schwachstellen in *ihrem* System zu identifizieren [Ale03, HMA04], helfen diese Ansätze ihnen nicht bei der Vermeidung von Flaws. Um die Anforderung A3 zu erfüllen modellieren Entwickler in MontiSecArc nur solche Elemente, die aus ihrer konstruktiven Sicht dazu beitragen, dass das System sicher ist. Daher werden insbesondere keine Angreifer, Details wie die verwendeten Verschlüsselungsverfahren und Security-Fachtermini in MontiSecArc spezifiziert.

Huiqun et al. [YHDM04] nennen mit Zugriffskontrolle, Security Level, und Informationsflusskontrolle bereits viele dieser wesentlichen Security-Eigenschaften, die eine Security-Architektur aus Entwicklersicht ausmacht, ohne jedoch eine Herleitung für diese Aussage oder eine Architekturbeschreibungssprache anzugeben, welche diese Konzepte beinhaltet.

Kapitel 5.

Security-Architekturanalyse mit MontiSecArc

Die Sprache MontiSecArc ermöglicht es die Security-Architektur eines Systems bereits vor der Implementierung zu modellieren. Allein durch die Abstraktion, welche MontiSecArc bereitstellt, werden Flaws jedoch nicht verhindert, da es mit MontiSecArc auch möglich ist Security-Architekturen zu modellieren, die Flaws enthalten. Dies ist beispielsweise bei der Modellierung bestehender Systeme notwendig, um bereits implementierte Sicherheitsmechanismen zu erfassen. Um jedoch Flaws in der Entwicklung zu verhindern, müssen MontiSecArc-Modelle überprüft werden, bevor sie zur Erstellung der Implementierung genutzt werden. Durch die präzise Modellierung von Security-Eigenschaften in MontiSecArc wird es möglich Security-Architekturmodelle anzugeben und automatisiert zu entscheiden, ob diese Flaws enthalten. Wie McGraw [McG06, S. 161] feststellt, werden Security-Architekturanalysen manuell, ad-hoc und von unterschiedlichen Personen durchgeführt. Daher sind diese Analysen nicht wiederholbar und ihre Ergebnisse auch nicht konsistent. Zusätzlich hat jeder Experte unterschiedlich ausgeprägte Erfahrungen, sodass diese auch zu unterschiedlichen Ergebnissen kommen. Testprozesse, Checklisten und Methoden wie STRIDE können das Problem lindern, jedoch bleibt ein wesentlicher Bestandteil: Die fehlende Automatisierung, welche für eine verlässliche, regelmäßige und schnelle Wiederholbarkeit notwendig ist. Durch eine Automatisierung dieser Analyse wird es ebenfalls möglich diese in einem agilen modellgetriebenen Entwicklungsprozess einzusetzen, wie dies in Anforderung A1 gefordert wird.

Auch der Microsoft SDL [HL06, S 165] empfiehlt die während der Design Phase erstellten Designdokumente anlässlich der Architekturanalyse in der Testphase auf den neusten Stand zu bringen und die Entwurfsdokumente an die Implementierung anzupassen. Wird MontiSecArc in einem agilen MDD Prozess eingesetzt, so werden solche Abweichungen des Modells von der Implementierung nicht auftreten, da die MontiSecArc-Modelle auch zur Generierung der Implementierung genutzt werden und somit einen Teil des Systems definieren.

Klassische Architecture Risk Analysis erfordert einen Security-Experten, der diese durchführt. Um Anforderung A3 zu erfüllen, müssen die Analysen so realisiert werden, dass diese die Destruktivität eines Angreifers mitbringen, Entwickler die Ausgaben der Analysen verstehen können und das modellierte System so anpassen können, dass die Analysen bei einer erneuten Anwendung keine Weaknesses mehr melden. Durch die Automatisierung werden die Fragen, welche Auditoren stellen wie beispielsweise “Ist diese Verbindung verschlüsselt?” bereits durch die Analyse behandelt, sodass sich die Auditoren auf komplexere Flaws konzentrieren können, die noch nicht automatisiert geprüft werden.

Aus typischen Fragen von Auditoren entsteht somit eine Sammlung von Flaw Correction Pattern, die über mehrere Systeme hinweg zur Identifikation von Flaws wiederverwendet werden kann. Hat sich eine solche Sammlung als Standard etabliert, kann diese genutzt werden, um nachzuweisen, dass die Architektur ein gewisses Maß an Security erfüllt (Anforderung A6), indem alle Flaw Correction Pattern auf die zu prüfende Architektur angewandt werden und das Ergebnis festgehalten wird. Dies ermöglicht eine automatisierte Zertifizierung gemäß eines durch Flaw Correction Pattern klar definierten Standards, den eine Architektur genau dann erfüllt, wenn keines der Flaw Correction Pattern einen Flaw meldet.

Dem Designprinzip Complete Mediation (Anforderung A4) folgend sollten Sprachen und Werkzeuge in der Softwareentwicklung sicherstellen, dass Schutzmaßnahmen überall dort in ein System eingebaut werden, wo es notwendig ist, sodass Softwareentwickler gar nicht erst in die Lage versetzt werden Systeme zu entwickeln, die Weaknesses enthalten. Durch Programmiersprachen wie Java wurde bereits gezeigt, dass Vulnerabilities wie Bufferoverflows vollständig verhindert werden können, wenn die vom Entwickler verwendete Sprache Konzepte nicht unterstützt, die bei einer Fehlbenutzung zu Vulnerabilities führen. In Java wird die fehleranfällige Aufgabe der Speicherverwaltung, die für die Mehrheit der heute bekannten Vulnerabilities verantwortlich ist, nicht in die Hände des Entwicklers gelegt, sondern automatisiert durch Compiler und Laufzeitumgebung realisiert. Somit sind Bufferoverflows in Java nicht möglich, solange Compiler und Laufzeitumgebung korrekt implementiert sind.

Dieser sprachbasierte Ansatz zur Verhinderung von Vulnerabilities wird auch bei MontiSecArc verfolgt. Hier werden Flaws allerdings nicht verhindert, indem Konzepte wie unverschlüsselte Konnektoren in der Sprache nicht vorkommen und durch die Laufzeitumgebung automatisch als verschlüsselte Verbindungen umgesetzt werden. Stattdessen werden MontiSecArc-Modelle in einer Analysephase auf bekannte Flaws untersucht und automatisiert bzw. durch den Entwickler korrigiert, bevor sie weiterverarbeitet werden. Diese direkte Korrektur von Fehlern durch den Entwickler wird durch den agilen modellgetriebenen Entwicklungsprozess möglich, in den die Analysen eingebettet sind, da die Analysen direktes Feedback für den Entwickler liefern. Durch diese frühe Erkennung von Flaws sind zum einen die Kosten zur Behebung minimal und zum anderen lernen Entwickler durch das direkte Feedback, wie sie einen Flaw in Zukunft vermeiden. Somit entstehen gemäß Anforderung A2 keine hohen Zusatzkosten durch eine Analyse.

Die in MontiSecArc als Sprachkonzepte vorhandenen Schutzmaßnahmen werden im Anschluss an diese Analyse durch einen Generator in eine Implementierung überführt. Somit werden Flaws auf Modellebene vor der Übersetzung in ausführbaren Code behandelt und sichergestellt, dass alle Flaws, auf die das Modell überprüft wurde, nicht in das fertige System gelangen, solange der vorgesehene agile modellgetriebe Entwicklungsprozess eingehalten wird.

Produkte, die im Softwareentwicklungsprozess automatisierte Security-Analysen durchführen, beginnen wie HP Fortify [Hew15] frühestens während der Implementierung mit der Statischen Analyse des GPL Quellcodes [CM04]. Somit helfen diese um Bugs wie Bufferoverflows und Injection Vulnerabilities aufzuspüren. Um mit einem solchen Ansatz Flaws zu finden, muss aus diesem Code allerdings zunächst die Architekturbeschreibung extrahiert werden, was für sich ein schwierige Aufgabe darstellt [DP09].

Wie bereits in Kapitel 2.2.2 beschrieben wird, automatisiert das Microsoft Threat Modeling

Tool [Mic15] die STRIDE Methode zur Security-Architekturanalyse. Der hier vorgestellte Ansatz zur Analyse von MontiSecArc-Modellen beruht im Gegensatz zu STRIDE nicht auf einer festen Menge von Analysen, sondern lässt sich durch Flaw Correction Pattern erweitern. Außerdem verfolgt MontiSecArc einen MDD Ansatz, sodass Modelle nach der Analyse auch zur Konstruktion des Systems verwendet werden, was die Inkonsistenz zwischen MontiSecArc-Modell und realisiertem System eleminiert. Die Teile von STRIDE, die sich in Flaw Correction Pattern ausdrücken lassen, werden in den folgenden Abschnitten vorgestellt. Analysen für Denial-of-Service und Elevation of Privilege werden jedoch nicht auf MontiSecArc-Modelle angewandt, da Denial-of-Service als Teil von Safety nicht im Fokus von MontiSecArc liegt und somit auch kein Sprachelement zur Verfügung steht, welches zufällige Fehler von Komponenten oder Konnektoren modelliert. Außerdem entsteht Elevation of Privilege im Kern durch Bugs bei der Verarbeitung von Daten oder in der Implementierung der Authentication und Authorization und nicht durch Flaws. Daher wird dies im Rahmen der Security-Architekturanalyse nicht weiter betrachtet.

Berger et al. [BSK16] übersetzen ein Extended Data Flow Diagram (EDFD) in einen Graphen und führen auf diesem Analysen durch, mit dem verschiedene in den CWE beschriebenen Flaws gefunden werden. Zur Beschreibung der Flaws wird eine Graph Query Language verwendet. Diese erlaubt es zwar auch Transformationen zu beschreiben um die gefundenen Flaws zu korrigieren, allerdings wird diee Möglichkeit nicht genutzt. Ein Grund dafür könnte sein, dass in EDFDs keine Schutzmaßnahmen definiert werden können, wodurch die Analysen auch viele Flaws identifizieren gegen die das System bereits geschützt ist.

Faily et al. [FLN+12] wenden bekannte Angriffe aus den CAPECs und CWEs als Angriffsmuster gegen ein Systemmodell an, welches Rechte und Rollen, sowie Komponenten und Konnektoren verwendet. Dabei wird jedoch keine Beschreibungssprache dieser Angriffsmuster vorgestellt.

Aus der Literaturanalyse von van den Berghe et al. [vdBSYJ15] geht hervor, dass lediglich UMLsec eine automatische Analyse von Security-Eigenschaften erlaubt, die über Zugriffskontrolle hinausgeht und ohne einen Security-Experten durchgeführt werden kann. Wie bereits in Kapitel 4 dargestellt wurde, ist UMLsec besonders auf Modelle, die Verhalten beschreiben, ausgerichtet. Daher liegt auch der Schwerpunkt der Analysen von UMLsec auf der Analyse von Protokollen und nicht auf der Component & Connector-Architekturanalyse. So lassen sich in UML Deployment Diagrammen mit UMLsec jedoch überprüfen, ob die Verbindungen die Confidentiality Anforderungen für das Angreifermodelle, welches auf dieser Verbindung definiert wurde, erfüllen [JSB08].

5.1. Modellierung von Flaw Correction Pattern

Im Folgenden wird zunächst beschrieben wie Flaw Correction Pattern modelliert werden, bevor bekannte Flaws auf diese Art formalisiert werden.

Da MontiSecArc es erlaubt die Security-Architektur eines Systems zu beschreiben, eignet sich die Sprache auch dafür Beispiele für Security-Architekturen anzugeben, die einen Flaw enthalten, sodass bekannte Flaws auf diese Art dokumentiert werden können. Dies ermöglicht es Flaw Correction Pattern genauso wie Design Pattern [GHJV94, AIS+77] zu dokumentieren. Ein Flaw Correction Pattern ist analog zur Definition eines Design Pattern definiert:

Definition 5.1.1 (Flaw Correction Pattern) *Ein Flaw Correction Pattern ist ein Lösung für einen wiederkehrenden Flaw, die diesen Flaw beseitigt.*

Die Beschreibung von Flaw Correction Pattern ist an sich ein Schritt vorwärts, da so überhaupt eine Diskussion über Flaw Correction Pattern möglich wird, an deren Ende eine Sammlung bekannter Flaw Correction Pattern steht. Entwickler können diese Pattern beim Entwurf sicherer Systeme nutzen, um darauf zu achten, dass die in den Flaw Correction Pattern beschriebenen Flaws nicht in der Architektur enthalten sind und ggf. die im Flaw Correction Pattern vorgeschlagene Lösung anwenden, um einen Flaw zu beseitigen.

Wie sich bei der Formalisierung bekannter Flaws gezeigt hat, lassen sich nicht alle Flaws als Flaw Correction Pattern beschreiben, da nicht alle Flaws eine eindeutige Lösung besitzen oder schon die Suche nach einem Flaw unzuverlässig ist. Wenn sich jedoch die Suche zumindest charakterisieren und analog zu Flaw Correction Pattern formalisieren lässt, wird dies als Flaw Smell bezeichnet, der in Anlehnung an einen Code Smell darauf hindeutet, dass ein Flaw vorliegen kann. Die Erkennung und Modellierung von Flaw Smells verläuft demnach analog zu Flaw Correction Pattern, mit dem Unterschied, dass ein Flaw Smell Entwicklern nur als Hinweis auf einen möglichen Flaw dient, der manuell behoben werden muss. Daher werden Flaw Smells im Folgenden zunächst analog zu Flaw Correction Pattern behandelt.

Die Komplexität einer detaillierten Systemarchitektur und die Menge an existierenden Flaw Correction Pattern wird jedoch höchstwahrscheinlich dazu führen, dass bei einer manuellen Suche nicht alle Flaws einer Architektur gefunden werden. Ein Weg die Effizienz der Suche zu verbessern, ist sie zu automatisieren, sodass ein Entwickler oder Security-Experte, der Flaws in einer Architektur sucht, durch automatisierte Architekturanalysen unterstützt wird. Um Flaws automatisiert zu suchen müssen diese in einer Sprache formuliert werden, die es ermöglicht Aussagen über MontiSecArc-Modelle zu formulieren. Aus der UML bietet sich hierzu die Object Constraint Language (OCL) [Obj14b] an, welche eine eigene Syntax besitzt, die unabhängig von der Modellierungssprache ist, über die Aussagen formuliert werden. Diese Unabhängigkeit hat den Vorteil, dass die OCL universell einsetzbar ist, bringt allerdings auch den Nachteil mit sich, dass die Syntax, in der die Bedingungen formuliert werden, keinen Bezug zur Security-Architektur haben. Daher wurde zur Beschreibung von Flaw Correction Pattern der Ansatz der domänenspezifischen Transformationssprachen [Vis02, BW07, Grø09] verwendet. In einer solchen Transformationssprache lässt sich beschreiben, wie ein Modell einer Sprache in ein anderes Modell derselben Sprache überführt wird. Domänenspezifische Transformationssprachen verwenden dabei zu großen Teilen die Syntax der zugrundeliegenden Modellierungssprache, also MontiSecArc im konkreten Fall und ermöglichen es so den Nutzern der Modellierungssprache auch die Transformationssprache zu nutzen, ohne viel neue Syntax zu lernen. Insbesondere ist auch ein in MontiSecArc modelliertes Beispiel für einen Flaw in dieser Transformationssprache gültig, sodass es als Ausgangspunkt für die Entwicklung eines Flaw Correction Pattern in der Transformationssprache genutzt werden kann. Durch die Transformation wird beschrieben, wie ein Modell, das einen Flaw enthält, transformiert wird in ein Modell, das diesen Flaw nicht besitzt. Somit wird die Erkennung eines Flaws zusammen mit seiner Lösung in einer Transformation beschrieben.

Diese Transformationen können automatisiert in einem agilen modellgetriebenen Entwicklungsprozess angewandt werden, sodass eine Security-Architekturanalyse durch Flaw Correction

Pattern ohne einen Security-Experten möglich ist (Anforderung A1). Security-Experten werden durch diese Automatisierung entlastet, da sie Flaws, die bereits in Flaw Correction Pattern formuliert sind und zuverlässig erkannt werden, nicht länger manuell identifizieren und korrigieren müssen. Somit können sie sich darauf konzentrieren die Sonderfälle, welche nicht von Flaw Correction Pattern abgedeckt werden, manuell zu analysieren oder diese ebenfalls in Flaw Correction Pattern zu übersetzen. Da die Entwickler direkt bei der Ausführung der Transformation die Ergebnisse der Analyse erhalten, wird ihnen auch das Wissen über Flaws vermittelt, die sie typischerweise machen und sie können diese in Zukunft bereits bei der Erstellung des Entwurfs vermeiden.

Durch die Arbeit von Hölldobler, Weisemöller et al. [HRW15a, Wei12, HHRW15] ist es möglich aus der MontiCore Grammatik einer Sprache eine dazugehörige domänenspezifische Transformationssprache abzuleiten und ein Verarbeitungswerkzeug zu generieren, welches Transformationen auf Modelle dieser Sprache anwendet.

```
1 package de.rwth.se.secarc.analysis;                                MSA^T
2
3 module ClientAuth {
4
5   transformation accessPort() {
6     SecArcComponent $C [[ component $client {
7       accesscontrol on;
8     } ]]
9
10    connect $client.$_ -> $server.$someInPort;
11
12    SecArcComponent $S [[component $server {
13      port in $someInPort;
14      [[ :- accesscontrol on; ]];
15    } ]]
16
17    where {
18      $C.getTrustlevel() < $S.getTrustlevel()
19    }
20  }
21
22  main(){
23    loop accessPort();
24  }
25 }
```

Listing 5.1: Eine MontiSecArc Transformation, die Authorization vom Client auf den Server verschiebt.

Listing 5.1 zeigt eine in dieser Sprache beschriebene Transformation, die das Flaw Correction Pattern *Client-Side Authentication* [HL02, S. 687] formalisiert, bei dem ein Client, der ein geringes Trust Level hat Zugriffskontrolle durchführt und ein Server, der ein höheres Trust Level

besitzt, sich auf diese Zugriffskontrolle verlässt. In diesem Fall kann ein Angreifer sich als Client ausgeben, direkt auf den Server zugreifen und so die Zugriffskontrolle umgehen. Eine Lösung für diesen Flaw ist die Zugriffskontrolle in der Komponente mit dem höheren Trust Level durchzuführen, also dem Server.

In Listing 5.1 wird im Matching-Teil der Transformation zuerst beschrieben, wann der Flaw in einem Modell vorhanden ist. Dazu wird in Zeile 6-8 die Existenz der Client Komponente und des Sprachelements `access` in der Komponente gefordert. Damit die Transformation für unterschiedliche Modelle gilt, werden Variablen eingeführt, die mit $ beginnen und für einen beliebigen Wert im Modell stehen. Wird eine Variable an mehreren Stellen in der Transformation verwendet, müssen im Modell all diese Stellen den selben Wert haben, damit der Matching-Teil der Transformation erfüllt ist. Hier wird eine Kombination aus abstrakter und konkreter Syntax von MontiSecArc verwendet, wobei als Konvention Elemente aus der abstrakten Syntax mit Großbuchstaben beginnen und die konkrete Syntax mit Kleinbuchstaben. Die konkrete Syntax wird hier in doppelten eckigen Klammern geschrieben, um sie von der Abstrakten zu unterscheiden.

Zeile 10 beschreibt die Verbindung zwischen Client und Server, wobei `$client` der zuvor beschriebenen Komponente entspricht und der Zielport `$someInPort` des Konnektors ein eingehender Port der Komponente `$server` ist. Der Quellport des Konnektors ist für das Flaw Correction Pattern unerheblich und wird daher mit der anonymen Variable `$_` entsprechend variabel markiert. Diese anonyme Variable verwirft den Wert jedoch wieder, sodass sie an verschiedenen Stellen der Transformation eingesetzt werden kann, an denen beliebige unterschiedliche Modellelemente stehen.

Zeile 12-15 beschreiben die Serverkomponente und in Zeile 14 wird durch den Ersetzungsoperator `:-` Zugriffskontrolle mittels `accesscontrol on` in diese Komponente eingefügt. Im Allgemeinen wird der Ausdruck links des Ersetzungsoperators mit dem Ausdruck auf der rechten Seite ersetzt. Da hier nur auf der rechten Seite etwas steht, wird dieser Ausdruck eingefügt.

Im `where`-Block in Zeile 17-19 werden Bedingungen in Java Syntax über die zuvor definierten Variablen formuliert, die zutreffen müssen, damit der Matching-Teil der Transformation erfüllt ist. Hier wird ausgedrückt, dass das Trust Level des Clients kleiner als das des Servers sein muss, wobei die Methode `getTrustlevel()` auf der abstrakten Syntax der Komponenten verwendet wird. Für Analysen, die über Pattern Matching hinausgehen, wie beispielsweise Pfadanalysen bietet der `where`-Block die Möglichkeit den AST des MontiSecArc-Modells an eine in Java geschriebene Analyse zu übergeben, die dann entscheidet, ob ein Flaw vorhanden ist.

In der Transformationssprache lässt sich also nicht nur der Flaw des Flaw Correction Pattern angeben, sondern auch direkt die Transformation, wie der Flaw behoben werden kann und in eine Lösung transformiert wird, die den Flaw nicht enthält. Somit können Security-Experten erstmals Flaw Correction Pattern präzise formalisieren, um darüber zu kommunizieren und in einer Diskussion die bekannten Flaw Correction Pattern zu identifizieren.

Der Ansatz von Almorsy et al. [AGI13], der ebenfalls auf die automatisierte Identifikation von Flaws ausgerichtet ist unterscheidet sich in mehreren Punkten von dem hier Vorgestellten. Zum einen verwenden Almorsy et al. die Security Domain-Specific Visual Language (SecDSVL), eine graphische DSL, die mehrere UML Diagrammarten erweitert, um die Security-Eigenschaften in der Architektur darzustellen und zum anderen verwendet der Ansatz OCL um Flaws zu

beschreiben. Wie bereits diskutiert, bietet die Notation eines Flaw Correction Pattern in einer domänenspezifischen Transformationssprache den Vorteil, dass Konzepte aus der Security-Architekturbeschreibungssprache direkt zur Beschreibung von Flaws verwendet werden können und außerdem eine Transformation hin zu einer Lösung ohne Flaw angegeben werden kann, wodurch das Flaw Correction Pattern vollständig modelliert wird.

5.2. Formalisierung bekannter Flaws als Flaw Correction Pattern

Um ein MontiSecArc-Modell automatisiert auf bekannte Flaws zu testen, müssen diese als Flaw Correction Pattern formalisiert werden. Je mehr Flaws durch eine automatische Security-Architekturanalyse identifiziert werden können, desto eher lohnt sich der Einsatz eines Analysewerkzeugs im Entwicklungsprozess, das diese prüft. Daher werden hier folgende Flaw Correction Pattern beschrieben, die erfolgreich formalisiert werden konnten und zur Nutzung in Analysewerkzeugen bereitstehen:

- *Untrusted Connector*
- *Vulnerabilities von Drittanbieterkomponenten*
- *Multiple Access Points*
- *Direct Access*
- *Client Tampering*
- *Authentication over Untrusted Connector*

Als Quelle für Flaws dienen dabei mehrere Sammlungen von Angriffstechniken, die bereits in Kapitel 2 vorgestellt wurden. Diese Angriffe wurden zunächst daraufhin untersucht, ob sie einen Bug oder einen Flaw ausnutzen. Als Flaw Correction Pattern wurden nur solche formalisiert, die einen Flaw in der Security-Architektur ausnutzen und nicht alleine durch einen Bug in der Implementierung möglich werden.

5.2.1. *Untrusted Connector*

Die unverschlüsselte Übertragung von Informationen ist nicht per se unsicher. Wenn diese Kommunikation innerhalb einer vertrauenswürdigen Komponente geschieht, in der kein Angreifer die Kommunikation beeinflussen kann, ist auch eine Verschlüsselung nicht notwendig. Ist jedoch die Komponente, welche die Kommunikationspartner enthält, weniger vertrauenswürdig ist als die Kommunikationspartner, handelt es sich um einen Flaw. Das Flaw Correction Pattern *Untrusted Connector* in Listing 5.2 formalisiert diesen Sachverhalt und fügt das Schlüsselwort `encrypted` in den Konnektor ein, um sicherzustellen, dass die Verbindung verschlüsselt wird. Damit entspricht dieses Flaw Correction Pattern dem Information Disclosure auf einem Konnektor aus der STRIDE Analyse.

```
1 package de.rwth.se.secarc.analysis;                                MSA^T
2
3 module UntrustedConnector {
4   transformation encryptUntrusted() {
5     SecArcComponent $Env [[ component $envirounment {
6       SecArcComponent $C [[ component $client {} ]]
7       connect [[ :- encrypted ]] $client.$_ -> $server.$_;
8       SecArcComponent $S [[ component $server {} ]]
9     } ]]
10
11    where {
12      $Env.getTrustlevel() < $C.getTrustlevel()
13        || $Env.getTrustlevel() < $S.getTrustlevel()
14    }
15  }
16  main(){
17    encryptUntrusted();
18  }
19 }
```

Listing 5.2: Eine MontiSecArc Transformation, die Konnektoren wo notwendig verschlüsselt.

5.2.2. *Vulnerabilities von Drittanbieterkomponenten*

Jeden Tag werden zahlreiche Vulnerabilities in Produkten veröffentlicht, um Nutzern dieser Produkte zu ermöglichen angemessen auf diese neu bekannt gewordenen Vulnerabilities zu reagieren. Die Reaktion von Nutzern einer Komponente beinhaltet zwei Schritte, die von MontiSecArc unterstützt werden. Zum einen muss überhaupt festgestellt werden, dass eine Komponente von einer bekannten Vulnerability betroffen ist und zum anderen sind die Auswirkungen einer erfolgreichen Ausnutzung dieser Vulnerability auf die Security-Architektur des Gesamtsystems zu betrachten.

Durch die Menge an bekannten Vulnerabilities ist es praktisch unmöglich alle Schwachstellen aller Komponenten eines Systems manuell nachzuverfolgen. Daher bauen Softwarehersteller automatische Update Mechanismen in ihre Produkte ein, die regelmäßig die aktuellste Version der Software über das Internet herunterladen und installieren. Dieses Verfahren ist zum einen nur für Komponenten geeignet, die über einen Internetzugang verfügen und zum anderen können fehlerhafte Updates einen Verlust der Verfügbarkeit einer Komponente und im schlimmsten Fall des gesamten Systems verursachen. Ein anderer Ansatz, der ursprünglich von Auditoren eingesetzt wurde, um Systeme auf bekannte Vulnerabilities zu testen, sind Vulnerability Scanner wie Nessus [BDM+04] oder OpenVAS [Ope16]. Diese Scanner werden zunehmend auch regelmäßig eingesetzt, um kontinuierlich die laufenden Systeme auf Vulnerabilities zu prüfen, die beispielsweise nicht durch ein automatisches Update behoben wurden.

Diese Scanner versuchen im ersten Schritt das laufende Softwareprodukt und seine Version zu identifizieren. Daraus leiten sie eine ungefähre CPE ab und ermitteln, welche Vulnerabilities das Produkt aufweisen könnte. Die automatische Bestimmung der CPE, also der Produktbezeichnung und seiner genauen Version, ist dabei besonders erfolgreich, wenn das Produkt je nach Versi-

onsnummer markante Unterschiede in der Interaktion mit dem Scanner aufweist. Besonders bei Frameworks und Bibliotheken, die dafür konzipiert sind um darauf aufbauend einen neuen Dienst zu realisieren, der einen hohen Grad an Individualität besitzen kann und somit keine markanten Merkmale aufweist, ist diese automatische Erkennung ungenau. Durch diese ungenaue Bestimmung der CPE werden alle Aussagen über vorhandene Vulnerabilities in einem System ebenfalls ungenau. Durch das Sprachelement `cpe` bietet MontiSecArc die Möglichkeit direkt die CPE des verwendeten Produkts anzugeben, welches eine Komponente realisiert. Somit wird die ungenaue Ermittlung der CPE umgangen und eine Analyse kann direkt die bekannten Vulnerabilities aus der NVD abrufen.

Für einzelne Komponenten können Vulnerability Scanner Vulnerabilities erkennen und automatische Updates können diese beheben, allerdings geben diese Methoden keine Aussage darüber, in wieweit eine Vulnerability die gesamte Security-Architektur beeinflusst. Hier kommt MontiSecArc zum tragen, welches es ermöglicht, die Vulnerabilities der Produkte im Kontext des Gesamtsystems zu analysieren. Die Folgen einer ausgenutzten Vulnerability in einer Komponente können vielfältig sein. In MontiSecArc wird für die Analyse davon ausgegangen, dass die betroffene Komponente vollständig kompromittiert wurde und unter Kontrolle eines Angreifers steht. Daher wird das Trust Level dieser Komponente im Modell auf -1 gesetzt, da der Angreifer von der übernommenen Komponente aus weitere Teile des Systems angreifen kann. Auf diesem Modell, welches die bekannten *Vulnerabilities von Drittanbieterkomponenten* berücksichtigt, werden wiederum alle Flaw Correction Pattern angewandt, um Flaws zu identifizieren, die durch die Vulnerabilities erst möglich werden. Diese Analyse wird während des Betriebs eines Systems dazu verwendet, um die Auswirkungen eines Angriffs abzuschätzen und Schutzmaßnahmen gegen eine Ausnutzung in der Architektur zu identifizieren.

5.2.3. Physische Security

Wie in Abschnitt 4.1.7 beschrieben, gibt es die drei Schutzmaßnahmen `shield`, `evident` und `sensor` in MontiSecArc um eine Komponente gegen physische Angriffe zu schützen. Welche Maßnahme für welche Komponente angemessen ist, liegt in der Hand der Entwickler. Ein Flaw liegt auf jeden Fall vor, wenn eine Komponente durch keine der möglichen Schutzmaßnahmen gegen einen mit `env` angegebenen Zugang zum System geschützt ist. In der durch MontiSecArc bereitgestellten Analyse werden Modelle auf diesen Flaw hin analysiert und solche Angriffe ausgegeben, die durch keine Gegenmaßnahme abgewehrt werden.

Je nach Art und Verwendungszweck des modellierten Systems ist es sinnvoll neben diesem grundlegenden Schutz, den diese Analyse sicherstellt, noch weitere Schutzmaßnahmen vorzusehen. Entwickler und Security-Experten definieren diese gemeinsam in einer Policy, die die mit MontiSecArc bereitgestellte Analyse erweitert. Diese Policy wird durch Flaw Correction Pattern beschrieben, die es auch erlauben Policies für konkrete Modelle zu formulieren, sodass beispielsweise für eine besonders schützenswerte Komponente sowohl `shield` als auch `evident` für einen Kanal vorhanden sein muss.

```
1 module SingleAccessPoint {                                          MSAT
2   transformation singleAccessPoint() {
3     SecArcComponent $AccessPoint [[ component $accessPoint {
4       Access $A;
5       [[ :- port in  $dstPort; ]]
6       [[ :- port out $dstPort; ]]
7     } ]]
8
9     connect $accessPoint.$_ -> $service.$_;
10
11    SecArcComponent $Service [[component $service {
12    } ]]
13
14    [[ connect $otherAP.$srcPort -> $service.$dstPort; :- ]]
15    [[ :- connect $otherAP.$srcPort -> $accessPoint.$dstPort; ]]
16    [[ :- connect $accessPoint.$dstPort -> $service.$dstPort; ]]
17
18    where {
19      $otherAP != $accessPoint;
20    }
21  }
22  main(){
23    loop singleAccessPoint();
24  }
25 }
```

Listing 5.3: Flaw Correction Pattern *Multiple Access Points*.

5.2.4. *Multiple Access Points*

Der Zugriffsschutz eines Systemes, das viele unterschiedliche Zugänge hat, ist allein durch die Anzahl der Stellen, an denen kontrolliert werden muss, ob ein Zugriff erlaubt ist, komplexer und damit anfällig für Fehler als ein System, bei dem es nur einen Zugang gibt, an dem Zugriffskontrolle durchgeführt wird. Das Design Pattern *Single Access Point* [YB97] fordert daher, dass es zu einem System immer nur einen Zugang gibt, der die Zugriffskontrolle überprüft.

Durch das Sprachkonzept von `accesscontrol on` oder `access` bietet MontiSecArc bereits die Möglichkeit für eine Komponente zu definieren, dass alle eingehenden Ports dieser Komponente eine Zugriffskontrolle durchführen. *Single Access Point* lässt sich für eine Komponente und ihre Subkomponenten daher bereits durch ein Sprachelement aus MontiSecArc darstellen.

Die eigentliche Intention des Security Pattern ist jedoch, dass es eine separate Komponente gibt, die vor dem Zugriff auf einen Service die Zugriffskontrolle durchführt, sodass der Service diese nicht durchführen muss. Dies widerspricht dem Prinzip *Be Reluctant to Trust*, da der Service darauf vertraut, dass der Single Access Point korrekt funktioniert und die Verbindung zu ihm ebenfalls vertrauenswürdig ist. Das Pattern dient daher zur besseren Strukturierung der Zuständigkeiten von Komponenten innerhalb eines Trust Levels. Ein Flaw in der Architektur, die

dieses Pattern umsetzt, ist demnach ein Konnektor, der am Single Access Point vorbei auf einen Service zugreift. Das Flaw Correction Pattern *Multiple Access Points* in Listing 5.3 beschreibt genau diese Situation und entfernt in Zeile 14 den Konnektor, welcher am AccessPoint (Zeile 4) vorbei auf einen Service zugreift, der über den AccessPoint erreichbar ist (Zeile 9). Als Lösung wird der entfernte Konnektor über zusätzliche Ports am AccessPoint (Zeile 5-6) wieder eingefügt (Zeile 15-16).

Die umfangreiche Literatur zu Security Design Pattern [YHSJ06, YWM08, SFBH+13] legt die Vermutung nahe, dass diese eine gute Quelle für Flaw Correction Pattern darstellen, indem die Nichtexistenz eines Security Design Patterns als Flaw gewertet wird. Tatsächlich bezieht sich ein großer Teil der Security Design Pattern jedoch auf die Implementierung von Sicherheitsmechanismen in objektorientierten Programmiersprachen und ist daher nur sehr begrenzt auf Component & Connector-Architekturen anwendbar. Außerdem ist das Fehlen eines Design Pattern kein eindeutiges Zeichen für einen Flaw, sondern nur ein Hinweis auf eine Verbesserungsmöglichkeit.

5.2.5. Security-Designprinzipien

Security-Designprinzipien (siehe Abschnitt 2.1.4) bieten allgemeine Hinweise, die beim Entwurf eines Systems beachtet werden sollen, damit es besonders widerstandsfähig gegen Angriffe ist. MontiSecArc unterstützt mit seinen Sprachelementen Entwickler dabei zu dokumentieren, ob gewisse Designprinzipien wie z.B. Compartmentalization befolgt und auch in der Implementierung konstruktiv umgesetzt werden. Ob ein MontiSecArc-Modell andere Designprinzipien wie Defense in Depth und Be Reluctant to Trust erfüllt, lässt sich, wie im Folgenden beschrieben wird, durch eine Analyse des Modells überprüfen.

Defense in Depth

MSA

```
1 package de.rwth.se.secarc.analysis;
2
3 component Lan {
4   trustlevel +1 "The Lan is protected by a Firewall";
5
6   component A a {
7     port out Data;
8   }
9
10  component B b {
11    port in Data;
12  }
13
14  connect a.data -> b.data;
15 }
```

Listing 5.4: Ein MontiSecArc-Modell, welches bei Anwendung der *Defense in Depth Analyse* einen *Untrusted Connector* Flaw besitzt.

Das Prinzip *Defense in Depth* besagt, dass mehre Schutzmaßnahmen, die sich ergänzen, in einem System verwendet werden sollen. Eine Architektur, die dieses Prinzip umsetzt toleriert (Konfigurations-)Fehler in einer Schutzmaßnahme, sodass das System alleine durch einen Fehler keine Vulnerability bekommt, sondern zumindest noch einen grundlegenden Schutz gegen Angriffe besitzt.

In MontiSecArc beschreibt das Trust Level zusammen mit den anderen Schutzmaßnahmen, wie ein System Angreifern den Zugang verweigert. Um zu untersuchen, in wieweit ein MontiSecArc-Architekturmodell das Prinzip *Defense in Depth* umsetzt, wird in dieser Analyse mit einer Ein-Fehler-Annahme überprüft, ob der Ausfall einer einzelnen Schutzmaßnahme die Sicherheit des Gesamtsystems beeinflusst. Dazu wird jeweils eine Schutzmaßnahme aus dem Modell entfernt und auf dieser geschwächten Architektur werden alle Analysen angewandt, um Flaws aufzuzeigen, die durch den Verlust einer Schutzmaßnahme entstehen. Da das Trust Level von nicht modellierten Schutzmaßnahmen abstrahiert, wird die Fehlfunktion dieser Maßnahmen simuliert, indem in dieser Analyse das Trust Level einzelner Komponenten auf das Trust Level ihrer Oberkomponente gesetzt wird, bzw. auf -1, wenn keine Oberkomponente existiert. Somit wird angenommen, dass die Schutzmaßnahmen, von denen das Trust Level abstrahiert überwunden wurden und die Komponente kompromittiert ist.

Das klassische Szenario in dem Defense in Depth angewandt wird, ist in Listing 5.4 dargestellt. Hier besteht das System exemplarisch aus zwei Komponenten, die miteinander kommunizieren und einer Oberkomponente, die ein höheres Trustlevel hat, weil eine Firewall das Netzwerk und somit all ihre Subkomponenten schützt. Fällt die Firewall aus oder wird sie falsch konfiguriert ist es einem Angreifer unter Umständen möglich den Netzwerkverkehr zwischen Komponenten A und B mitzulesen und zu verändern. Wird in der *Defense in Depth Analyse* dieses Modells das Trust Level entfernt, erkennt das Flaw Correction Pattern *Untrusted Connector*, dass der Konnektor zwischen A und B unverschlüsselt und somit verwundbar ist.

Diese Analyse erkennt Flaws, die behoben werden müssen, damit die Architektur das Defense in Depth Prinzip mit einer Ein-Fehler-Annahme vollkommen erfüllt. Da bei einer Security-Architektur, die Defense in Depth nicht erfüllt, sehr viele Änderungen notwendig werden können, um das Prinzip zu erfüllen, kann die Anzahl an Flaws, die bei dieser Analyse identifiziert wurden, als Metrik für den Erfüllungsgrad des Prinzips herangezogen werden. Somit können bei Architekturänderungen verschiedene Varianten bezüglich des Defense in Depth Prinzips verglichen werden und eine schrittweise Verbesserung der Architektur wird möglich. Da jedoch das Messen von Security eine sehr komplexe Aufgabe ist [PC10], bietet die *Defense in Depth Analyse* lediglich die Möglichkeit die einzelnen Flaws unterschiedlich zu gewichten. Die Beurteilung, ob eine Architektur, die die *Defense in Depth Analyse* nicht vollständig erfüllt, angemessen sicher ist, wird dem Auditor oder Entwickler überlassen.

Reluctance to Trust

Die Verarbeitung von Daten findet typischerweise über mehrere Komponenten verteilt statt. Im Beispiel in Listing 5.5 überprüft Komponente A die Eingaben, die sie erhält beispielsweise auf einen gültigen Wertebereich (keine negativen Preise) und sendet diese Daten an Komponente B. B könnte nun darauf vertrauen, dass alle empfangenen Daten immer von A überprüft wurden. Durch diese Annahme entsteht ein Großteil aller Flaws, denn ein Angreifer hat in dieser Situation

MSA

```
1 package de.rwth.se.secarc.analysis;
2
3 component Env {
4   component A a {
5     port out Data;
6   }
7   component B b {
8     trustlevel +1;
9     port in Data;
10  }
11  connect a.data -> b.data; //Flaw
12 }
```

Listing 5.5: In dieser Architektur verlässt sich Komponente B auf die korrekte Verarbeitung in einer Komponente A.

die folgenden Möglichkeiten, die Prüfung von A zu umgehen:

1. Er kann die Nachrichten bei der Übertragung verändern, wenn ein `unencrypted connector` durch eine Komponente mit geringem Trust Level führt.

2. Er kann Nachrichten direkt an B schicken, wenn die Oberkomponente von B ein geringeres Trust Level hat als B.

3. Er kann den Programmablauf von A ändern, wenn er Zugriff auf die Komponente A hat, und A ein geringeres Trust Level hat als B.

Daher besagt sowohl das Security-Designprinzip *Reluctance to Trust* als auch das Pattern Authorative Source of Data [Rom01] sowie STRIDE, welches dies als Spoofing und Tampering bezeichnet, dass eine Komponente nicht darauf vertrauen darf, dass die Daten wohlgeformt sind und A diese überprüft hat. Vergleichbar ist auch der Ansatz des Defensive Programming, bei dem jede Methode annimmt, dass sie absichtlich mit falschen Parametern aufgerufen wird und daher alle Parameter vor der Benutzung überprüfen muss (beispielsweise auf nicht erlaubte null-Referenzen) [Blo08, S. 181].

Es gibt Schutzmaßnahmen in der Security-Architektur, die die Ausnutzung dieser Flaws erschweren, indem der Zugriff beispielsweise auf eine Nutzergruppe eingeschränkt wird, aber vollständig verhindern lässt er sich nur, indem vor der Verarbeitung der Nachricht diese auf syntaktische Korrektheit überprüft wird [BDL+14] und anschließend gemäß dem Defensive Programming überprüft wird, ob der Inhalt der Nachricht für den Empfänger semantisch sinnvoll ist, bevor er darauf reagiert.

Im Rahmen der *Reluctance to Trust Analyse* lassen sich die drei zuvor beschriebenen Fälle von Flaws identifizieren. Auch eine Beschränkung des Zugriffs auf weniger Nutzer lässt sich in der Architektur durch die Einführung von Zugriffskontrolle und Authentication realisieren, eine vollständige Lösung ist jedoch nur unter Berücksichtigung der vollständigen Implementierung

```
1 package de.rwth.se.secarc.analysis;                                MSA^T
2
3 module ReluctanceToTrust {
4   transformation directAccess(){
5     SecArcComponent $Env [[ component $_ {
6       connect $client.$_ -> $server.$in;
7       SecArcComponent $S [[component $server {
8               [[ :- access $in (Policy_$client);]]
9       }]]
10    } ]]
11    where {
12      $Env.getTrustlevel() < $S.getTrustlevel()
13    }
14  }
15
16  main(){
17    directAccess();
18  }
19 }
```

Listing 5.6: Das Flaw Correction Pattern Direct Access fügt `accesscontrol on` hinzu.

möglich, da hierzu alle Überprüfungen aus A erneut in B durchgeführt werden müssen[1].

Betrachtet man die drei beschriebenen Möglichkeiten, wie ein Angreifer die Prüfung in A umgehen könnte, fällt auf, dass die erste Variante bereits durch das Flaw Correction Pattern *Untrusted Connector* behandelt wird.

Der zweite Angriff lässt sich durch den Einsatz von Authentication und Authorization auf einen bestimmten Nutzerkreis begrenzen. Das Flaw Correction Pattern *Direct Access* in Listing 5.6 fügt hierzu Zugriffskontrolle für Ports von Komponenten hinzu, die Ziel eines Konnektors sind, der durch ein geringeres Trust Level als das der Komponente verläuft. Dies ist jedoch nur ein Teil der Lösung, da nicht automatisch entschieden werden kann, wie der für die Zugriffskontrolle notwendige `identity` Link eingefügt werden soll. Dieses Problem wird in Abschnitt 5.2.6 detaillierter untersucht. Durch diese vom Flaw Correction Pattern *Direct Access* bereitgestellte Lösung kann nur ein legitimierter Nutzer des Systems den Angriff durchführen. Bei Systemen mit einem offenen Nutzerkreis, die über eine Selbstregistrierung verfügen, oder, wenn unzufriedene Angestellte zu Innentätern werden, hilft diese Schutzmaßnahme nicht.

Der dritte Fall, in dem der Angreifer die Ausführung in A manipuliert, kann mit dem Flaw Correction Pattern *Client Tampering* in Listing 5.7 nur erkannt werden. Eine Lösung wäre eine Authentication die B nachweist, dass A unverändert ist und beispielsweise über Hardwareschutzmaßnahmen eine Kompromittierung verhindert. Das Trust Level abstrahiert von diesen Details, daher spiegelt sich ein solcher erhöhter physischer Schutz in einem erhöhten Trust Level wider. Daher identifiziert das Flaw Correction Pattern durch einen Vergleich der Trust Level der

[1] In Kapitel 6.2 wird ein Ansatz vorgestellt, der Entwickler bei der Implementierung dazu anhalten soll diese Überprüfung der Daten durchzuführen.

```
1 package de.rwth.se.secarc.analysis;                                MSA^T
2
3 module ReluctanceToTrust {
4
5   transformation clientTampering(){
6
7     SecArcComponent $C [[component $client {  }]]
8     connect $client.$_ -> $server.$checkInput;
9     SecArcComponent $S [[component $server {  }]]
10
11    where {
12      $C.getTrustlevel() < $S.getTrustlevel()
13    }
14  }
15
16  main(){
17    clientTampering();
18  }
19 }
```

Listing 5.7: Das Flaw Correction Pattern *Client Tampering* identifiziert Ports als $checkInput, an denen Daten erneut überprüft werden müssen.

Komponenten `$C` und `$S` (Zeile 12) den Port `$checkInput` in Zeile 8, an dem in der Implementierung Daten beim Empfang auf ihre syntaktische und semantische Korrektheit überprüft werden müssen.

5.2.6. `identity` Link und Encrypted Connector

Durch einen `identity` Link wird die Authentizität des Nutzers zwischen zwei Komponenten sichergestellt, wenn diese Komponenten keine gemeinsame Oberkomponente haben, die vertrauenswürdig ist. Kommunizieren die Komponenten, zwischen denen der `identity` Link eingesetzt wird, über unverschlüsselte Verbindungen, kann ein Angreifer die übertragenen Nachrichten auf diesen Verbindungen verändern. Somit ist es dem Angreifer möglich Nachrichten im Namen eines anderen Nutzers an eine Komponente zu senden und dadurch die Authentication und Authorization zu umgehen, da zwar eine sichere Authentication eingesetzt wird, aber die eigentliche Kommunikation der Komponenten nicht gegen Veränderung geschützt ist. Um diesen Flaw zu verhindern, wird das in Listing 5.8 dargestellte Flaw Correction Pattern *Authentication over Untrusted Connector* angewendet, welches die Konnektoren, die parallel zu einem `identity` Link verlaufen, als `encrypted` markiert. Hierbei werden zwei Transformationen eingesetzt, um `strong` und `weak identity` Links zu berücksichtigen.

```
package de.rwth.se.secarc.analysis;                                    MSA^T

module IdentityEncryption {

  transformation encryptStrongIdentity() {
    component $client { }
    identity strong $client -> $server;
    connect [[ :- encrypted ]] $client.$_ -> $server.$_;
    component $server { }
  }

  transformation encryptWeakIdentity() {
      component $client { }
      identity weak $client -> $server;
      connect [[ :- encrypted ]] $client.$_ -> $server.$_;
      component $server { }
    }

  main(){
    encryptStrongIdentity();
    encryptWeakIdentity();
  }

}
```

Listing 5.8: Dieses Flaw Correction Pattern *Authentication over Untrusted Connector* fügt `encrypted` zu Konnektoren hinzu, damit die Authentication nicht umgangen werden kann.

5.2.7. Analyseergebnisse priorisieren

Wird die Security-Architektur eines bestehenden Systems erstmals mit MontiSecArc analysiert, werden unter Umständen sehr viele Flaws identifiziert, sodass nicht klar ist, welcher Flaw zuerst behoben werden soll. In einer solchen Situation wird eine Priorisierung der Flaws nach Schwere oder Kritikalität im Kontext des Systems benötigt, damit die gravierendsten Flaws zuerst behoben werden können. MontiSecArc bietet zur Priorisierung während der Modellierung das Sprachelement `critical` Port an, welches die wichtigsten Funktionen eines Systems identifiziert.

Diese Information wird nun genutzt, um die Security-Architektur auf ihren wichtigen Kern zu reduzieren, indem alle Pfade von Konnektoren, die über `critical` Ports führen, gebildet werden. Alle Komponenten, Ports und Konnektoren, die Teil dieser Pfade sind, zusammen mit den Schutzmaßnahmen, die sie enthalten, bleiben in der reduzierten Security-Architektur erhalten. Alle anderen Elemente werden aus dem MontiSecArc-Modell entfernt und diese reduzierte Security-Architektur wird auf Flaws mit den bereits beschriebenen Pattern analysiert. Durch diese Einschränkung werden nur Flaws identifiziert, die die wichtigen Teile des Systems betreffen und somit zuerst behoben werden sollten.

5.3. Flaw Smells

Neben Flaw Correction Pattern, bei denen zu einem Flaw eine eindeutige Lösung direkt mit angegeben wird, gibt es auch Situationen, in denen ein Flaw nicht eindeutig automatisch gelöst werden kann, oder schon die Identifikation des Flaw nicht zuverlässig funktioniert. Solche Fälle werden als Flaw Smell bezeichnet, in Anlehnung an Code Smells, die auf Bugs im Quellcode hindeuten. Folgende Flaw Smells werden hier beschrieben, die automatisiert in MontiSecArc-Modellen identifiziert werden können:

- *Identity Access Smell*
- *Identity Trustlevel Smell*
- *Unencrypted Connector Smell*
- *Ports Without Access Control*

5.3.1. `identity` Link und Access

Wird am Zielport eines Konnektors Authorization durchgeführt, muss es auch eine Authentication zwischen den Komponenten geben, die der Konnektor verbindet, damit die Identität der Nutzer vor der Authorization eindeutig festgestellt werden kann. In Listing 5.9 ist ein Beispiel dargestellt, welches auf den ersten Blick zu diesem *Identity Access Smell* passt, da es keinen `identity` Link zwischen den Komponenten A und B gibt. Allerdings wird hier ein Identity Provider (IdP) verwendet, über den die Authentication zwischen A und B abläuft.

Bei der *Identity Access Smell* Analyse wird dieser Fall berücksichtigt, indem nur `identity` Links als fehlend angemerkt werden, wenn es keinen Pfad von `identity` Links zwischen zwei Komponenten gibt, die durch einen Konnektor verbunden sind, der auf einem Port mit Zugriffskontrolle endet. Daher wird für den Konnektor zwischen A und C ein fehlender `identity` Link gemeldet, jedoch nicht zwischen A und B. Eine automatische Korrektur dieses Flaws ist allerdings nicht ohne weiteres möglich, da der Einsatz von IdPs eine Entwurfsentscheidung ist und diese daher nicht verlässlich in der Architektur erkannt werden können, wenn sie bereits vorhanden sind. Außerdem muss der Entwickler entscheiden, ob `weak`- oder `strong` `identity` Links eingesetzt werden sollen.

Genauso wie Zugriffskontrolle einen `identity` Link bedingt, gilt auch die umgekehrte Beziehung. Wenn zwei Komponenten mit einem `identity` Link verbunden sind, muss die Zielkomponente zumindest einen Port besitzen, der über eine Zugriffskontrolle verfügt, sonst wäre der `identity` Link überflüssig. In Listing 5.10 tritt dieser *Identity Access Smell* zwischen den Komponente A und B auf.

Betrachtet man hier den Fall des IdP ergibt sich das Problem, dass der IdP, als Ziel eines `identity` Link, keine Zugriffskontrolle verwendet und somit eine fehlende Zugriffskontrolle beim IdP gemeldet wird. Die Bedingung zu lockern und lediglich zu fordern, dass das Ende eines Pfads von `identity` Links Zugriffskontrolle durchführen soll, ist nicht zielführend wie das Beispiel zeigt, da durch diese gelockerte Bedingung die fehlende Authorization bei B nicht identifiziert würde.

MSA

```
1 package de.rwth.se.secarc.analysis;
2
3 component Env {
4   component A a {
5     port out Data;
6   }
7
8   connect a.data -> b.data;
9
10  component B b {
11    accesscontrol on;
12    port in Data;
13  }
14
15  component IdP idp {
16    trustlevel +1;
17  }
18
19  identity weak a -> idp;
20  identity weak idp -> b;
21
22  component C c {
23    accesscontrol on;
24    port in Data;
25  }
26  connect a.data -> c.data;
27 }
```

Listing 5.9: In diesem Beispiel werden fälschlicherweise die Flaw Smells *Identity Access Smell* und *Identity Trustlevel Smell* erkannt.

Da eine fehlende Zugriffskontrolle bedeutet, dass jeder Nutzer Zugriff auf einen Port besitzt und ein fehlender `identity` Link, dass niemand Zugriff erhält, weil die Zugriffskontrolle keine authentischen Nutzerdaten erhält, wird durch die *Identity Access Smell* Analyse sichergestellt, dass keine Zugriffskontrolle vergessen wurde und eine falsch-positiv Meldung des IdP in Kauf genommen.

5.3.2. `identity` Link und Trust Level

Ein `identiy` Link von Komponente A zu Komponente B dient dazu der Zielkomponente B die Identität von Nutzern nachzuweisen, um Zugriff auf Ports von B zu erhalten. B ist i.d.R. besser gegen Angreifer geschützt als A, sodass es ungewöhnlich ist, wenn A ein höheres Trust Level besitzt, da ein Angreifer in diesem Fall direkt B angreifen würde und der `identity` Link überflüssig wäre. Daher meldet die Analyse *Identity Trustlevel Smell* `identity` Links, bei denen die Quellkomponente ein höheres Trust Level als die Zielkomponente besitzt.

Betrachtet man jedoch das Beispiel in Listing 5.9, in dem ein Identity Provider (IdP) eingesetzt

MSA

```
1 package de.rwth.se.secarc.analysis;
2
3 component Env {
4   component A a {
5     port out Data;
6   }
7
8   component B b {
9     port in Data indata;
10    port out Data outdata;
11  }
12
13  component C c {
14    accesscontrol on;
15    port in Data;
16  }
17
18  connect a.data -> b.indata;
19  connect b.outdata -> c.data;
20
21  component IdP idp { }
22  identity weak   a -> b;
23  identity weak   b -> idp;
24  identity weak idp -> c;
25 }
```

Listing 5.10: In diesem Beispiel verwendet B ein Backend C und führt keine Zugriffskontrolle durch, obwohl A sich bei B authentisiert.

wird, der ein höheres Trust Level als A und B hat, zeigt sich, dass der `identity` Link vom IdP zu B nicht von dem zuvor beschriebenen Flaw Smell zu unterscheiden ist. Daher liefert diese Analyse beim Einsatz eines IdP falsch-positiv Ergebnisse.

5.3.3. Verschlüsselte Pfade

Unverschlüsselte Übertragung von Informationen kann nicht nur bei der Betrachtung einer direkten Verbindung ein Problem sein, sondern auch auf einem Pfad, der über mehrere Konnektoren hinweg Informationen transportiert. Wird wie im Beispiel in Listing 5.11 die Verbindung von a1 zu b1 und b2 zu b3 verschlüsselt, obwohl das Trust Level der Oberkomponente dies nicht erforderlich erscheinen lässt, werden über diese Verbindung anscheinend besonders schützenswerte Informationen übertragen. Sind weitere Verbindungen auf dem Pfad, zu dem diese Verbindung gehört, unverschlüsselt, ist dies zumindest auffällig und könnte darauf hindeuten, dass dort vergessen wurde Verschlüsselung einzusetzen.

Daher listet die *Unencrypted Connector Smell* Analyse alle Pfade auf, die mindestens eine verschlüsselte Verbindungen enthalten und mindestens eine unverschlüsselte und markiert Letztere. Im Beispiel in Listing 5.11 werden die Verbindungen von a1 zu b1 und b2 zu b3 als Flaw

```
1 package de.rwth.se.secarc.analysis;                    MSA
2
3 component Env {
4   trustlevel +1;
5
6   component A a1 {
7     port out Data;
8   }
9
10  component B {
11    port in Data inData;
12    port out Data outData;
13  }
14  component B b1, b2, b3;
15
16  component C c1 {
17    port in Data;
18  }
19
20  connect a1.data -> b1.inData;
21  connect encrypted b1.outData -> b2.inData;
22
23  connect b2.outData -> b3.inData;
24  connect encrypted b3.outData -> c1.data;
25 }
```

Listing 5.11: Beispiel für einen Pfad mit unverschlüsselten Teilen.

markiert, da sie auf einem Pfad von a1 zu c1 liegen, der verschlüsselte Verbindungen enthält. Der Entwickler oder Auditor erhält durch diese Analyse schnell einen Überblick darüber, ob es Teilpfade gibt, an denen ein Angreifer eine Übertragung beeinflussen könnte und daher der Konnektor verschlüsselt werden sollte.

Eine ähnliche Analyse wird von Im und McGregor für AADL beschrieben [IM09]. Dabei werden die erlaubten Datenflüsse zusammen mit den Security Levels der Komponenten, auf die sie lesend oder schreibend zugreifen dürfen, in einem Szenario beschrieben. Anhand dieses Szenarios wird das AADL Modell daraufhin überprüft, ob diese Datenflüsse auch höchstens auf diese Security Level zugreifen.

5.3.4. Konventionen

Drittanbieterkomponenten werden durch eine Konfiguration an das System angepasst. Diese Konfiguration, die mit dem Schlüsselwort `configuration` in einer Komponente benannt wird, beinhaltet oftmals Einstellungen, die bei unsachgemäßer Konfiguration zu Vulnerabilities führen. Im OWASP Testing Guide zielt der Punkt *Configuration Management Testing* genau auf diese Konfiguration ab, sodass Konfigurationen vor ihrem Einsatz überprüft werden sollten. Als Konvention wird einer solchen überprüften Konfiguration der Suffix `_reviewd` gegeben, sodass

Konfigurationen, die noch nicht überprüft wurden, leichter zu identifizieren sind und durch eine Analyse gesammelt ausgegeben werden können.

5.3.5. Effektive Zugriffsrechte

Wird durch die Zugriffskontrolle der Zugriff für bestimmte Ports auf einzelne Policies beschränkt, so stellt sich die Frage, welche Zugriffsrechte Nutzer, die diese Policy erfüllen, im gesamten System besitzen. Diese Frage leitet sich aus den Designprinzipien Least Privilege und Separation of Privilege her, die fordern Nutzern nur die notwendigen Zugriffsrechte zu gewähren und wo notwendig Berechtigungen zur Durchführung einer Aufgabe auf mehrere Personen aufzuteilen. Um diese Fragen beantworten zu können, wird eine View bereitgestellt, die darstellt, auf welche Ports und Komponenten eine Policy Zugriff hat. Somit wird die Darstellung der Zugriffskontrolle aus dem MontiSecArc-Modell umgekehrt, um eine bessere Übersichtlichkeit für die Analyse der Designprinzipien Least Privilege und Separation of Privilege bereitzustellen.

Buyens et al. [BSJ11] untersuchen, ob das Designprinzip Least Privilege für die Vergabe von Berechtigungen in einer Architektur eingehalten wurde, indem sie drei Typen von Verletzungen des Prinzips identifizieren und diese durch Transformationen der Architektur und der Berechtigungen beheben. Für die Analyse wird dabei das Architekturmodell mit Zugriffsrechten angereichert und Aufgaben, die im System erfüllt werden sollen, werden an die Konnektoren annotiert, welche zur Erfüllung der Aufgabe notwendig sind. Auf Basis dieser Aufgabenbeschreibung wird geprüft, ob die Aufgaben auch mit geringeren Berechtigungen durchgeführt werden können. Durch die von Bayens et al. vorgestellten Veränderung der Zugriffs-Policy und Transformation der Architektur, lässt sich ein System so verändern, dass es das Least Privilege Designprinzip erfüllt. MontiSecArc besitzt kein Sprachelement, welches den Zusammenhang zwischen mehreren Konnektoren im Sinne einer übergreifenden Aufgabe modelliert, allerdings lässt sich eine Zugriffs-Policy stets so gestalten, dass pro Aufgabe genau eine Policy existiert, die ein Nutzer erfüllen muss. Durch diese vereinfachte Modellierung lassen sich die automatisierten Least Privilege Transformationen ebenfalls auf MontiSecArc anwenden.

5.3.6. Ports ohne Zugriffskontrolle

Das Security-Designprinzip Complete Mediation besagt, dass überall dort, wo ein Zugriff stattfindet, auch geprüft werden muss, ob dieser erlaubt ist. Ports eines MontiSecArc-Modells, an denen keine Zugriffskontrolle definiert ist, widersprechen diesem Designprinzip und werden daher in der Smell Analyse *Ports Without Access Control* ausgegeben, damit Entwickler und Auditoren überprüfen können, ob diese Ports wirklich ohne Zugriffskontrolle nutzbar sein sollen.

In MontiSecArc wird Nachweisbarkeit von Aktionen durch die Implementierung der Zugriffskontrolle realisiert. Die STRIDE Bedrohung Repudiation wird daher überall dort, wo Zugriffskontrolle stattfindet abgewehrt, sodass die Analyse *Ports Without Access Control* solche Ports auflistet, wo Repudiation noch möglich ist.

5.3.7. Unterstützung einer manuellen Security-Architekturanalyse

Um eine manuelle Security-Architekturanalyse in Form einer Risikoanalyse durchzuführen zu können, benötigt ein Auditor Kenntnis über Funktionen oder Informationen eines Systems, die das Ziel eines Angreifer sind. Diese Güter werden in MontiSecArc als `critical` Ports modelliert, die über die Analyse *List Critical Ports* dem Auditor bereitgestellt werden. Somit kann der Auditor auch ohne genaue Kenntnis der Funktion des Systems sich auf die Teile konzentrieren, die die `critical` Ports beeinflussen können.

Kapitel 6.

Modellgetriebene Entwicklung von Systemen mit MontiSecArc

Dem agilen MDD Ansatz folgend (Anforderung A1) dienen MontiSecArc-Modelle nicht nur zur Security Architekturmodellierung und Dokumentation sowie zur frühen Analyse und Vermeidung von Flaws, sondern werden auch konstruktiv zur Erstellung der Implementierung eingesetzt. Um eine zuverlässige und einheitliche Übersetzung vom Modell zur Implementierung zu gewährleisten werden in diesem Kapitel Codegeneratoren, die auf Basis von MontiCore entwickelt wurden vorgestellt. Der zuerst vorgestellte Generator erzeugt die grundlegende Ausführungs- und Kommunikationsinfrastruktur, die es ermöglicht das Verhalten der Komponenten, die Nachrichten erhalten und versenden in Java zu implementieren. Wobei die Vernetzung der Komponenten, deren parallele Ausführung, die Authentication, Authorization und Verschlüsselte Kommunikation durch die generierte Infrastruktur realisiert wird. Eine erste Version dieses Generators wurde im Rahmen der Bachelorarbeit von Lucian Poth [Pot15] erstellt. Der als zweites vorgestellte Generator erzeugt Regeln für ein Network Intrusion Detection System (NIDS), welches die Netzwerkpakete überwacht, die die Nachrichten zwischen Komponenten übertragen. Diese Regeln werden aus den Konnektoren des MontiSecArc-Modell abgeleitet und definieren welche Verbindungen im Netzwerk erlaubt sind. Beobachtet das NIDS abseits der im Modell vorgesehenen Kommunikation weitere Pakete werden diese als Angriff gewertet und protokolliert. Die NIDS Regeln werden passend zur generierten Implementierung erzeugt, sodass dieses direkt zur Überwachung des Systems genutzt werden kann. Die grundlegende Modellanalyse, auf die dieser Generator aufbaut, zusammen mit Management Funktionen, die es ermöglichen ein verteiltes NIDS durch eine zentrale Webschnittstelle mit Regeln zu konfigurieren, wurde von Steffen Gräber [Gre15] implementiert. Diese Webapplikation erlaubt außerdem die Regeln aus einem MontiSecArc-Modellen abzuleiten und diese an die im Netzwerk verteilten Sensoren des NIDS zu verteilen und erkannte Angriffe zentral zu speichern und anzuzeigen.

6.1. Security-Implikationen der Modellgetriebenen Entwicklung

Es gibt zahllose Bibliotheken und Frameworks, die Schutzmaßnahmen wir Authentication und Authorization unterstützen und zur Umsetzung der Sprachelemente von MontiSecArc geeignet sind. Diese jedoch korrekt zu nutzen, sodass bei der Implementierung eines Systems keine Weaknesses entstehen ist eine komplexe Aufgabe. Die Schwierigkeit dabei ist, dass Entwickler explizit

die Schutzmaßnahmen in die Applikation einbauen bzw. diese aktivieren müssen. Vulnerabilities entstehen somit häufig indem eine dieser Schutzmaßnahmen in der Implementierung vergessen wurde. Um Anforderungen A3 und A4 zu erfüllen und den Entwickler von der Aufgabe zu befreien Schutzmaßnahmen, die bereits in der Architektur spezifiziert wurden, händisch korrekt zu implementieren, wird in diesem Kapitel ein Codegenerator vorgestellt, der die Modellelemente aus MontiSecArc-Modellen in Java Code übersetzen. Dieser Ansatz ermöglicht es im Sinne des agilen MDD MontiSecArc-Modell und Implementierung konsistent zu halten und somit sicherzustellen, dass Schutzmaßnahmen auch tatsächlich überall dort im System umgesetzt werden wo sie in der Architektur vorgesehen sind.

Somit kann die statische Analyse von Flaws, wie sie in Kapitel 5 beschrieben wird, auf MontiSecArc-Modellen durchgeführt werden und die Ergebnisse der Analyse gelten gleichermaßen für das Modell und die daraus generierte Implementierung. Im Rahmen eines Code Reviews, bei dem ein Auditor nach Weaknesses sucht muss die generierte Implementierung daher nicht auf Flaws untersucht werden, die bereits durch Flaw Correction Pattern im MontiSecArc-Modell behandelt wurden. Somit bietet der MDD Ansatz hier eine Abstraktion von der Implementierung, die es erleichtert Flaws zu identifizieren und zu beheben.

Das Modell ist natürlich nur eine zuverlässige Abstraktion der generierten Implementierung, solange der Generator keinen Code generiert, der Bugs enthält. Solche generierten Bugs treten schematisch auf und ihre Wirkung gleicht denen von Flaws, die ebenfalls dazu führen, dass ein System Vulnerabilities an vielen Stellen aufweist. Ein neu implementierter Generator muss daher gründlicher als die Implementierung des Verhaltens der Komponenten auf Bugs überprüft werden. Wird dieser Generator bei der Erstellung vieler Systeme verwendet und jedes mal begutachtet kehrt sich dies um, da der Generator mit der Zeit bessere Qualitätseigenschaften erhält und die Implementierung des Verhaltens der Komponenten mehr Vulnerabilities enthält als der häufig geprüfte Generator.

Da jede Umsetzung einer Schutzmaßnahme im Generator in einem Template definiert ist und diese Implementierung an jeder Stelle wo die Schutzmaßnahme angewandt wird eingesetzt wird kann ein Audit der Umsetzung dieser Maßnahmen sich auf die Templates des Generators konzentrieren. Wird im Gegensatz dazu in einem nicht MDD-Prozess kein Generator verwendet, muss ein Auditor alle Stellen im Code suchen, an denen die Schutzmaßnahme eingesetzt werden sollte und sicherstellen, dass dies auch korrekt umgesetzt wird. Durch die Verwendung des Generators konzentriert sich also die Arbeit bei einem Audit der Schutzfunktionen auf den Generator, der die Schutzmaßnahmen über eine gleichbleibende Schnittstelle mit der handgeschriebenen Implementierung verbindet und erspart fehleranfällige Suchen, die versuchen alle Stellen im Code zu identifizieren, an denen eine Schutzmaßnahme angewandt werden muss. Außerdem ist bei häufiger Wiederverwendung des Generators der Aufwand für ein einfaches Review geringer als bei einer händischen Implementierung Umsetzung der Schutzmaßnahmen, da jede dieser Implementierungen überprüft werden muss, der Generator jedoch nur einmal.

Durch den MDD Prozess, in dem die in MontiSecArc-Modellen definierten Schutzmaßnahmen direkt von einem Generator in Code übersetzt werden, können Auditoren sich bei der Analyse von MontiSecArc-Modellen auf Flaws konzentrieren und beim Review des Generators und der Implementierung der Komponenten auf Bugs.

Die Klassifizierung von van den Berghe et al. [vdBSYJ15] zeigt, dass 8 von 14 Ansätzen,

die ausschließlich Zugriffskontrolle modellieren, daraus auch eine Implementierung generieren. Exemplarisch kann hier SecureUML [LBD02] genannt werden, als erster Ansatz, bei dem Klassendiagramme mit Stereotypen um Rollen erweitert wurden und daraus Java Code für J2EE Applikationsserver generiert wird der den Applikationsserver anweist diese Rollen bei jedem Zugriff zu prüfen. Im Folgenden werden solche Ansätze diskutiert, die mehr als nur Zugriffskontrolle aus Modellen generieren.

Hoisl et al. [HSS14], Memon et al. [MMD+14] sowie Nakamura et al. [NTIO05, SNO06] generieren aus verschiedenen Modellierungssprachen WS-Policy [OAS12] Dokumente, die definieren, welche konkreten Algorithmen zum Schutz von Confidentiality und Integrity angewandt werden sollen.

Der Ansatz von Fernández-Medina et al. [FMP05, FMTVP07, TSFMP09, VBFMM12] ermöglicht es Berechtigungen und Security Level von Benutzern einer Datenbank bzw. eines Data Warehouse bereits im Datenmodell, welches durch ein Klassendiagramm mit Stereotypen realisiert ist, zu spezifizieren. Durch eine semi-automatische Übersetzung dieser Modelle in Schemata für Oracle 10g werden die Zugriffskontrolle und das Audit Loging von Oracle verwendet um die im Modell definierten Regeln durchzusetzen. Der hier vorgestellte MontiSecArc-Ansatz kann verwendet werden, um eine vollständige verteilte Applikation zu erstellen wobei die Authentication, Authorization sowie auch die verschlüsselte und integritätsgesicherte Übertragung der Nachrichten generiert wird, die nicht auf Webservices beschränkt ist.

6.2. Übersetzung von MontiSecArc in ein verteilt ausgeführtes System

Um Anforderung A2 zu erfüllen wurde MontiSecArc als general-purpose Modellierungssprache entwickelt, in der Security Schutzmaßnahmen unabhängig von der verwendeten Technologie darstellen werden. Durch den Einsatz von MontiCore lassen sich MontiSecArc-Modelle in einem agilen MDD Prozess zuerst auf bekannte Weaknesses prüfen und anschließend mittels templatebasierter Codegenerierung in die Realisierung des Systems übersetzen. Somit ist die Erstellung des MontiSecArc-Modells kein zusätzlicher Schritt, sondern dieser ersetzt die manuelle Implementierung von Zugriffskontrolle und verschlüsselter Übertragung von Nachrichten.

Bei der Implementierung eines Codegenerators müssen Technologieentscheidungen getroffen werden, um die abstrahierten Konzepte aus der Modellierungssprache mit einer konkreten Technologie zu realisieren. In diesem Abschnitt wird ein solcher Generator vorgestellt und die dazugehörigen Technologieentscheidungen erläutert. Dies ist jedoch nicht die einzige mögliche Realisierung, sondern andere Technologien, welche dieselben Schutzmaßnahmen realisieren können ebenfalls verwendet werden. Der hier vorgestellte Generator dient vor allem dazu zu veranschaulichen, wie MontiSecArc-Modelle sich konstruktiv sinnvoll im modellgetriebenen Entwicklungsprozess einsetzen lassen.

Atomare Komponenten kapseln Verhalten und Daten, die in der Architektur nicht spezifiziert werden. Dieses Verhalten wird in einer GPL beschrieben, um dem Entwickler die Möglichkeit zu bieten beliebige Funktionalität zu implementieren. Hier wird Java als GPL genutzt, um die

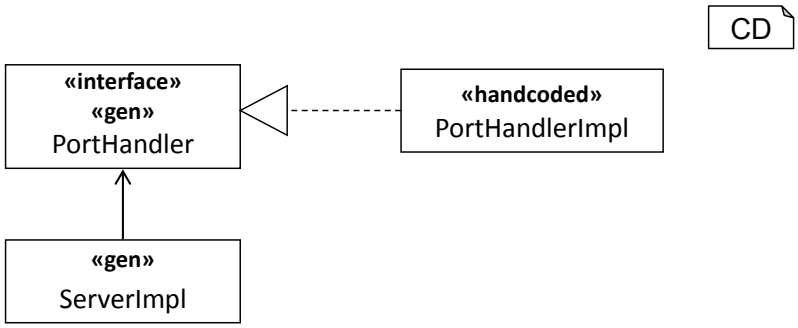


Abbildung 6.1.: Integration von handgeschriebenem Code in den generierten Server.

Integration von generiertem und handgeschriebenem Code über objektorientierte Mechanismen zu realisieren [GHK+15]. Der handgeschriebene GPL Code der Implementierung einer Komponente muss natürlich die in der Architektur spezifizierte Schnittstelle der Komponente erfüllen. Um dies zu gewährleisten wird aus jeder atomaren Komponente eine ServerImpl-Klasse generiert, die eine GPL Implementierung für die eingehenden Ports der Komponente erwartet und eine Client-Klasse, mit der die Implementierung Nachrichten über die in der Architektur vorgesehen ausgehenden Ports verschicken kann. Für die Integration des handgeschriebenen Codes in den Server wird Delegation eingesetzt, wie in Abbildung 6.1 dargestellt. Dabei implementiert die handgeschriebene Implementierung das für jeden eingehenden Port generierte Interface PortHandler. Diese handgeschriebenen PortHandlerImpl werden beim Server registriert, sodass der Server Nachrichten, die auf einem Port ankommen an den für diesen Port zuständigen PortHandlerImpl weiterleitet.

Jede Komponente läuft als eigener Prozess im Betriebssystem und kann somit jeweils auch unter einem anderen Betriebssystemaccount laufen. Somit hat sie nur Zugriff auf die Ressourcen die das Betriebssystem dem Account gewährt. Die einzelnen Prozesse können somit durch Mittel des Betriebssystems weitestgehend voneinander getrennt werden, sodass die Designprinzipien Least Privilege und Compartmentalization unterstützt werden.

Zur Realisierung der Konnektoren wird das Transmission Control Protocol (TCP) [Pos81] verwendet, welches einen bidirektionalen Kommunikationskanal realisiert, der zwischen einem Client TCP-Port und einem Server TCP-Port verläuft und durch den Client aufgebaut wird. Ein ausgehender Port in einem MontiSecArc-Modell entspricht dem Client TCP-Port und ein eingehender Port einem Server TCP-Port. Um konform zu den gerichteten Ports im Modell und dessen zugrundeliegender Semantik zu bleiben werden hier nur Nachrichten vom Client zum Server übertragen. Verwendet eine Implementierung also die generierte Client Klasse um eine Nachricht zu verschicken wird eine TCP Verbindung zum Server aufgebaut, die Nachricht wird übertragen und der Server delegiert die Verarbeitung der Nachricht an die entsprechende PortHandlerImpl.

Obgleich MontiSecArc-Modelle Informationen über Trust Level und auch physische Schutzmaßnahmen enthalten definieren Komponenten in einem MontiSecArc-Modell nicht auf welchem System in einem Netzwerk eine Komponente ausgeführt wird und daraus resultierend unter

welcher IP-Adresse ein Server erreichbar ist. Im Internet ermöglicht das Domain Name System (DNS) [Moc87] als zentraler Dienst die Übersetzung von Domain Namen zu IP-Adressen. Diese Zuordnung lässt sich nutzen, um die Verteilung der Komponenten eines MontiSecArc-Modells auf IP-Adressen zu beschreiben. Domain Namen sind genauso wie die voll qualifizierten Namen von Komponenten in MontiSecArc hierarchisch aufgebaut und durch Punkte getrennt.

MSA

```
1 package de.rwth.se.secarc;
2
3 component A {
4   component B {
5     component C d;
6     port out UserData;
7   }
8
9   connect encrypted d.userData -> e.userData;
10
11  component E {
12    port in UserData;
13    port in AdminData;
14  }
15
16  connect f.adminData -> e.adminData;
17  identity weak f -> e;
18
19  component F {
20      port out AdminData;
21  }
22 }
```

Listing 6.2: MontiSecArc Beispielarchitektur.

Die Komponente in Zeile 4 von Listing 6.2 hat den Namen `de.rwth.se.secarc.a.b` und die Komponente aus Zeile 5 `de.rwth.se.secarc.a.b.d`. Invertiert man diese Namen komponentenweise, so ergeben sich die Domain Namen `b.a.secarc.se.rwth.de` und `d.b.a.secarc.se.rwth.de`, für die im DNS die IP-Adressen hinterlegt werden unter denen diese Komponenten erreichbar sind. Da im DNS einzelne Domains und Sub-Domains wie z.B. `se.rwth.de` und `secarc.se.rwth.de` durch unterschiedliche Personen oder Organisationen verwaltet werden können lässt sich über das Package eines MontiSecArc-Modells steuern wer die Zuordnung von Komponente zu IP-Adresse vornimmt. Neben dieser globalen Namensvergabe lassen sich auch in der `hosts`-Datei des Betriebssystems Domain Namen eintragen, die dann allerdings nur innerhalb des Betriebssystems gültig sind.

In diese grundlegende Infrastruktur, die zur verteilten Ausführung von MontiSecArc-Modellen benötigt wird, wird als erstes MontiSecArc spezifisches Sprachelement der `encrypted` Konnektor übersetzt. Das Schlüsselwort `encrypted` weist den Generator an statt einer TCP Verbindung eine mit Transport Layer Security (TLS) [DR08] verschlüsselte Verbindung zu verwenden. TLS

ist gegen einen Man-in-the-Middle-Angriffe anfällig, wenn nicht mindestens ein Kommunikationspartner die Identität des Anderen überprüft. Bei diesem Angriff manipuliert ein Angreifer die Kommunikation so, dass die Kommunikationspartner denken sie würden direkt verschlüsselt miteinander kommunizieren, aber in Wirklichkeit beide mit dem Angreifer kommunizieren der zu beiden eine separate verschlüsselte Verbindung besitzt und so den Kommunikationsinhalt mitlesen und verändern kann. Zum Identitätsnachweis in TLS präsentiert ein Kommunikationsteilnehmer ein Zertifikat welches kryptographisch nachweist, dass der Host einen bestimmten Domain Namen hat. Der andere Kommunikationspartner überprüft das Zertifikat und bricht die Verbindung ab, wenn dieses als ungültig erkannt wurde. Da der Angreifer kein gültiges Zertifikat besitzt kann er zu einem der Kommunikationspartner keine Verbindung aufbauen und somit keinen Man-in-the-Middle-Angriff durchführen.

Dieses Verfahren kann sowohl der Client als auch der Server anwenden, auch wenn in der Praxis häufig der Client die Identität des Servers überprüft. Da durch das MontiSecArc-Modell festgelegt ist welche Clients auf einen Port eines Servers zugreifen kann der Server jedoch zusätzlich die Identität des Clients überprüfen und somit sicherstellen, dass auch nur die im Modell vorgesehenen Clients Zugriff auf den Server haben.

Damit gültige Zertifikate durch die generierte Implementierung von solchen unterschieden werden können, die ein Angreifer erstellt hat wird eine Public Key Infrastructure (PKI) benötigt in der eine Certificate Authority (CA) die Echtheit der Zertifikate bestätigt. Die Verwaltung und insbesondere die Erneuerung von Zertifikaten, die manuell durchgeführt wird, stellt einen erheblichen Aufwand für den Betrieb eines Systems darstellt. Die automatische Generierung und Verteilung von Zertifikaten für eine PKI geht über die Generierung einer Implementierung aus MontiSecArc-Modellen hinaus. Vielversprechend in diesem Bereich ist die Initiative Let's Encrypt [Int15], die es einem Server erlaubt automatisiert Zertifikate von einer anerkannten CA zu erhalten.

Die Vergabe von Zertifikaten an Benutzer zur Authentication hat immer noch keine weite Verbreitung gefunden. Daher unterstützt MontiSecArc die Authentication mit Benutzername und Passwort. Dabei wird nicht das Passwort selbst übertragen, sondern ein Schlüssel, der mit PBKDF2 [Kal00] deterministisch aus dem Passwort abgeleitet wurde. Somit verlässt das Passwort des Nutzers nicht den Client und der Server kann trotzdem feststellen, ob der Nutzer das richtige Passwort eingegeben hat. PBKDF2 verwendet eine Hashfunktion, die nur schwer invertierbar ist daher kann ein Angreifer der durch eine Vulnerability den Server kompromittiert und dort Zugriff auf den gespeicherten Schlüssel erlangt daraus nicht direkt sondern nur mit erheblichem Aufwand die Passwörter der Nutzer rekonstruieren, sodass die Nutzer im Falle einer Kompromittierung des Servers noch Zeit haben ihre Passwörter zu ändern bevor der Angreifer diese erraten hat und es für den Zugriff auf andere Systeme nutzen kann. Der Einsatz der Authentication und von PBKDF2 ist für den Entwickler des Verhaltens der Komponente transparent und kann daher auch nicht vergessen werden.

Genauso wird die im Modell spezifizierte Zugriffskontrolle durch den generierten Code behandelt bevor die Verhaltensimplementierung eines Servers aufgerufen wird. Als Zugriffskontrollschema verwendet die generierte Implementierung Role Based Access Control (RBAC) und nutzt die im Modell mit `access` spezifizierte Policy als Rollenname. Die Zugriffskontrolle stellt sicher, dass nur authentisierten Nutzern, die die für einen Port angegebene Rolle besitzen, der

Zugriff auf diesen Port gestattet wird. Um Fehler in der Implementierung der Zugriffskontrolle durch komplexe Ausnahmen für Ports ohne `access` Policy zu verhindern, wird für solche Ports durch die Zugriffskontrolle geprüft, ob der Nutzer die Rolle `noRoleDefined` besitzt. Der Benutzer `defaultUser` erhält diese Rolle und wird vom generierten Client verwendet um Nachrichten an Ports zu senden, für die keine `access` Policy spezifiziert ist. Für alle anderen Ports muss der Entwickler beim senden einer Nachricht einen Benutzernamen und ein Passwort übergeben, damit diese in der Authentication verwendet werden.

6.3. Generierung eines Network Intrusion Detection Systems

Neben Prävention von Angriffen durch Schutzmaßnahmen gibt es auch die Möglichkeit Angriffe während ihrer Vorbereitung beispielsweise durch Aktivitäten der aktiven Informationsgewinnung wie einen Portscan oder während eines Angriffs beispielsweise durch den Versand von Nachrichten, die nicht zum normalen Verhalten des Systems gehören zu beobachten. Ein Network Intrusion Detection System (NIDS) [Roe99], [Eck14, S. 178] überwacht den Netzwerkverkehr auf der Suche nach solchen Aktivitäten und dokumentiert diese. Leitet das NIDS automatisch Gegenmaßnahmen gegen den Angriff ein spricht man von einem Network Intrusion Prevention System (NIPS), bei dem ein Fehlalarm zur Folge hat, dass einem legitimen Nutzer der Zugang zum System verwehrt wird. Kann ein Angreifer einen solchen Fehlalarm gezielt herbeiführen, so lässt sich dieser zu einem Denial of Service (DoS) Angriff ausnutzen, bei dem der vom System erbrachte Service den Nutzern nicht zur Verfügung steht. Somit schadet ein falsch konfiguriertes NIPS einem System mehr als es nutzt. Im Folgenden wird diese Konfiguration zur Erkennung von Angriffen genauer betrachtet, die sowohl für NIDS als auch NIPS benötigt wird, daher wird im Folgenden ausschließlich der Begriff NIDS benutzt, obgleich die Konfiguration der Angriffserkennung auch für NIPS anwendbar ist.

Zur Erkennung von Angriffen verwendet ein NIDS Regeln die, ähnlich wie ein Virenscanner, beschreiben wie ein Netzwerkpaket aussieht, dass zu einem Angriff gehört. Somit wird es möglich Angriffe die bekannte Vulnerabilities ausnutzen anhand von bestimmten Inhalten im Netzwerkverkehr zu identifizieren. Dies erfordert jedoch, dass für jede Vulnerability eine Regel geschrieben werden muss, die jede Variation des Angriffs erkennt und diese in die Konfiguration jedes NIDS eingepflegt werden muss. Dies ist eine fortwehende Aufgabe die niemals vollständig erledigt werden kann, da täglich neue Vulnerabilities gefunden werden. Außerdem bewies Cohen bereits 1987, dass das erkennen eines Virus unentscheidbar ist [Coh87], was für NIDS bedeutet, dass auch die Erkennung von Schadcode, der in Angriffen auf Remote Code Execution Vulnerabilities eingesetzt wird nicht zuverlässig erkannt werden kann.

Um die Erkennung von Angriffen zu verbessern wird hier daher kein Blacklist Ansatz verwendet in dem versucht wird alle Angriffe aufzulisten, sondern ein Whitelist Ansatz, bei dem alle erlaubten Netzwerkaktivitäten beschrieben werden und Abweichungen davon als Angriff gewertet werden. Zur Erstellung dieser Whitelist werden die Informationen über Komponenten, Ports und Konnektoren aus dem MontiSecArc-Modell herangezogen. Durch die im vorherigen Abschnitt 6.2 beschrieben Technologieentscheidungen, werden diese Modellelemente in IP-Adressen und TCP-Ports übersetzt. Somit werden alle TCP-Verbindungen, die den Konnektoren des Modells entsprechen in die Whitelist des NIDS eingetragen und daher nicht als Angriff

erkannt.

Hier wird das NIDS Snort [Cis15, Roe99] verwendet, welches diese Whitelist anwendet. Die Konfiguration von Snort wird in mehrere Dateien aufgeteilt, sodass die generierte Whitelist in ein Snort integriert werden kann, welches mehrere Systeme überwacht. Außerdem kann die generierte Whitelist somit durch handgeschriebene Snort-Konfigurationen auf die Netzwerkumgebung angepasst werden.

6.4. Implementierung der Generatoren

Die zuvor beschriebenen Konzepte wie MontiSecArc-Modelle in eine Kommunikationsinfrastruktur und ein NIDS übersetzt werden sind in zwei Generatoren implementiert, die jeweils einen der Aspekte realisieren. Als Basis für diese Generatoren dient MontiCore, welches über das MontiCore DSL-Tool Maven Plugin aus einem Maven Build heraus aufgerufen wird. Der Maven Build wird durch einzelne Projekte aufgeteilt in den Generator, die Runtime und ein Anwendungsprojekt. Im Generator wird die Übersetzung von MontiSecArc-Modellen in ausführbaren Code mit Hilfe von FreeMarker Templates beschrieben. Die Runtime bündelt Code, welcher vom generierten Code zur Ausführungszeit benötigt wird. Im Anwendungsprojekt wird der Generator auf die dort gespeicherten MontiSecArc-Modelle angewandt und der generierte Code als Resultat des Projekts gespeichert. Im Folgenden wird zunächst der Generatoren für die Kommunikationsinfrastruktur und im Anschluss der Generator für das NIDS näher erläutert. Die Kommunikationsinfrastruktur bietet die Möglichkeit das Verhalten der Komponenten, welches nicht in MontiSecArc spezifiziert wird, in Java zu implementieren. Die Kommunikation zwischen den Komponenten wird dabei von der bereitgestellten Infrastruktur realisiert.

Java

```java
package de.rwth.se.secarc;

import java.io.IOException;
import java.net.Socket;
import de.rwth.se.secarc.constants.PortMapImpl;
import de.rwth.se.secarc.runtime.Client;

public class C_Client extends Client {

  public final Socket connect_UserData() throws IOException {
    PortMapImpl pm = new PortMapImpl();
    return openTLSConnection("de.rwth.se.secarc.e", pm.getPort("
        de_rwth_se_secarc_e_userData"));
  }
}
```

Listing 6.3: Generierter Client Code für die Komponente C aus Listing 6.2.

Das Klassendiagramm in Abbildung 6.4 zeigt einen detaillierten Überblick dieser Infrastruktur. Dabei bilden alle Klassen bis auf die am unteren Rand die Runtime. Die abstrakte Klasse `Client`, welche Teil der Runtime ist implementiert den Verbindungsaufbau von unverschlüsselten und

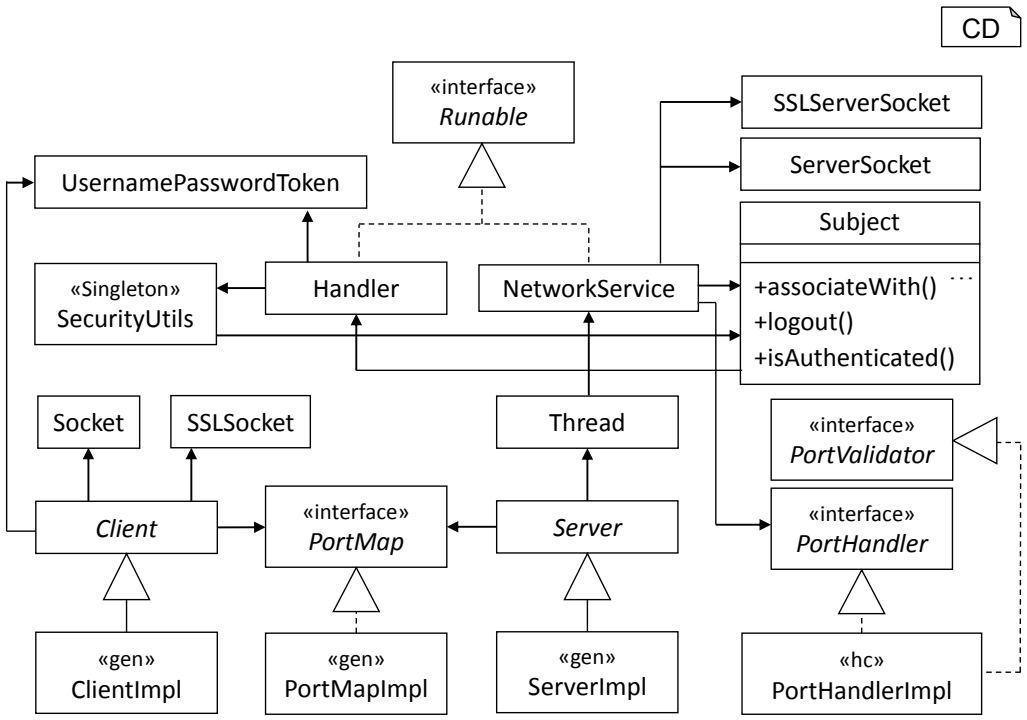


Abbildung 6.4.: Architektur der Java Implementierung.

verschlüsselten Verbindungen. Dazu wird die Klasse `SSLSocket` bzw. `Socket` aus der Java Runtime verwendet.

Für jede Komponente wird eine `ClientImpl` Klasse generiert, deren Name mit dem Komponentennamen beginnen und mit der Endung `_Client` endet. Listing 6.3 zeigt die Client Klasse für die Komponente `C` aus der in Listing 6.2 dargestellten Beispielarchitektur. Diese Klasse bietet für jeden ausgehenden Port eine `connect_` Methode, welche eine Verbindung zu dem eingehenden Port aufbaut der die Nachricht laut Modell empfangen soll. Die generierte Implementierung nutzt die Methode `openConnection` (Listing 6.5) bzw. `openTLSConnection` (Listing 6.3) der abstrakten Klasse `Client`, um eine unverschlüsselte bzw. verschlüsselte Verbindung aufzubauen. Somit ist für den Nutzer der `ClientImpl` Klassen transparent welche Art von Verschlüsselung verwendet wird und immer dann wenn im Modell definiert ist, dass eine verschlüsselte Verbindung verwendet werden soll wird die selbe Implementierung genutzt.

Die Übersetzung von Ports aus dem MontiSecArc-Modell in TCP Ports wird durch eine Klasse realisiert, welche das Interface `PortMap` implementiert. Aus dem Modell wird eine Standardimplementierung `PortMapImpl` generiert, bei der die Ports in aufsteigender Reihenfolge vergeben werden. Dies Implementierung kann auch durch den Nutzer der Kommunikationsinfrastruktur realisiert werden, um Ports aus dem Modell spezifischen TCP Ports zuzuweisen.

Nachrichten an eingehenden Ports einer Komponente werden asynchron bearbeitet. Daher verwendet die abstrakte Klasse `Server`, welche Teil der Runtime ist, einen `Thread` der wiederum einen `NetworkService` verwendet, um auf eingehende Verbindungen zu warten. `Net-`

`workService` implementiert das Interface `Runable`, sodass die Funktionalität der Klasse nebenläufig in dem Thread ausgeführt wird. Dabei bietet die `Server` Klasse analog zur Klasse `Client` die Möglichkeit auf unverschlüsselte oder verschlüsselte Verbindungen zu warten. Die unverschlüsselten Verbindungen verwenden die Klasse `ServerSocket` wohingegen für verschlüsselte Verbindungen die Klasse `SSLServerSocket` aus der Java Runtime verwendet wird, die eine TLS Implementierung bereitstellt. Beim Aufbau der TLS Verbindung prüfen Client und Servers gegenseitig ihre Zertifikate, um sicherzustellen dass sie mit genau der Komponente kommunizieren mit der sie erwarten zu kommunizieren und kein Man-in-the-Middle die Kommunikation beeinflusst. Analog zur Klasse `ClientImpl` wird für jede Komponente eine `ServerImpl` Klasse generiert, deren Name mit dem Komponentennamen beginnt. Diese nutzt die Runtime Implementierung aus der `Server` Klasse, um auf den im Modell spezifizierten Ports auf Verbindungen zu warten.

Wenn `NetworkService` eine eingehende Verbindung entgegengenommen hat, wird für alle Verbindungen für die eine `identity` Link im Modell definiert ist zunächst der Benutzer authentisiert. Im Anschluss wird über das Interface `PortHandler` das handgeschriebene Verhalten der Komponente aufgerufen, welches dieses Interface implementiert. Zur Authentication und Authorization wird dabei die RBAC Bibliothek Apache Shiro [Apa15b] verwendet. Shiro unterstützen mehrere Datenbanken und Identity Management Systeme, in denen Nutzer und ihre Rollen verwaltet werden. Da die Komponenten grundsätzlich verteilt ausgeführt werden, besitzt jede Komponente eine eigene Nutzerdatenbank. Durch die von Shiro unterstützten Identity Management Systeme kann jedoch auch eine gemeinsame Datenbank verwendet werden. Um die Implementierung auch ohne ein komplexes Identity Management System testen zu können unterstützt Shiro die Verwendung einer Konfigurationsdatei in der Benutzernamen, ihre Rollen und Passwörter gespeichert werden. Für jede Komponente wird eine solche Shiro Konfigurationsdateien generiert, in der alle Rollen, die an dieser Komponente genutzt werden, definiert werden. Entwickler können in dieser Datei Benutzer anlegen, die diese Rollen inne haben.

Zur Authorization in einer Anwendung bietet Shiro eine einfache Schnittstelle, welches in der Runtime wie folgt verwendet wird. Wenn `NetworkService` eine Verbindung annimmt, wird diese im Session Kontext einer neuen Instanz von `Subject` durch die Methode `associateWith()` registriert. In der nachfolgenden Bearbeitung der Verbindung prüft `Handler` in einem weiteren Thread über das Shiro Singleton `SecurityUtils` welches `Subject` in der Session aktiv ist. Durch die Methode `isAuthenticated()` wird anschließend geprüft, ob der Benutzer bereits authentisiert wurde. Bei einer Verbindung die neu aufgebaut wird ist dies nie der Fall. Daher sendet der Client das Password, welches der `Handler` verwendet, um einen von Shiro bereitgestellten `UsernamePasswordToken` zu erzeugen. Dieser Token wird von Shiro verwendet, um den Benutzer zu authentisieren. Diese Authentication und Authorization wird für jede Verbindung durchgeführt, bevor die handgeschriebene Implementierung aufgerufen wird. Somit kann ein Angreifer, der eine Komponente kompromittiert Zugriff auf alle Ports erlangen, für die keine `access` Policy definiert ist, wenn die Komponenten durch das Betriebssystem oder die verteilte Ausführung vollständig voneinander getrennt sind.

Für den Entwickler des handgeschriebenen Verhaltens bleibt ein Großteil dieser Details verborgen. Für ihn zeigt sich lediglich der Zusammenhang zwischen einem im Modell definierten `identity` Link und dem daraus resultierenden Interface des generierten Clients. So erfordert

```java
package de.rwth.se.secarc;

import java.io.IOException;
import java.net.Socket;
import de.rwth.se.secarc.constants.PortMapImpl;
import de.rwth.se.secarc.runtime.Client;

public class F_Client extends Client {

  public final Socket connect_AdminData(String inName, String
      inPassword) throws IOException {
    PortMapImpl pm = new PortMapImpl();
    return openConnection("de.rwth.se.secarc.e", pm.getPort("
        de_rwth_se_secarc_e_adminData"), inName, inPassword);
  }
}
```

Listing 6.5: Generierter Client Code für die Komponente F aus Listing 6.2.

der Client für F, welcher in Listing 6.5 dargestellt ist, aufgrund des in Listing 6.2 definierten `identity` Link zwischen F und E, dass beim Aufbau der Verbindung Benutzername und Passwort mit übergeben werden.

Gemäß dem Designprinzip Reluctance to Trust kann eine Komponente empfangen Daten nicht vertrauen, wenn diese aus einem Trust Level stammen, dass geringer ist als das der Komponente. Wie bereits in Abschnitt 5.2.5 beschrieben wurde lassen sich solche Situationen identifizieren und lösen indem alle Daten, die von einem Client mit geringerem Trust Level stammen vor ihrer ersten Nutzung überprüft werden. Diese Prüfung ist notwendig, da die Komponente nicht darauf vertrauen kann, dass alle Prüfungen, die der Client durchführen sollte auch tatsächlich durchgeführt wurden, da ein Angreifer sich als Client ausgeben könnte und somit die Prüfungen im Client umgehen kann. Dies wird auch als Security Pattern Client Input Filters [KETEH06] bezeichnet, welches im Rahmen der Beschreibung des Verhaltens einer Komponente umgesetzt werden muss. Um den Entwickler des Verhaltens darauf hinzuweisen, dass eine Nachricht überprüft werden muss bevor das Verhalten darauf reagiert wird für Ports, denen eine Komponente nicht vertrauen kann statt eines `PortHandler` Interfaces ein `PortValidator` Interface verwendet. Dieses Interface definiert eine Methode `validate`, die die Nachrichten entgegennimmt und eine Methode `handle` ohne Parameter. Die Implementierung dieses Interfaces soll die Nachrichten in der Methode `validate` überprüfen und in `handle` verarbeiten und wird daher in dieser Reihenfolge vom generierten Server aufgerufen. Wird diese vorgesehene Überprüfung nicht durchgeführt lässt sich dies in einem Codereview leicht feststellen indem die implementierten `validate` Methoden begutachtet werden.

Die Generierung des NIDS erfolgt in einem weiteren Generator, der ebenfalls in einem separaten Maven Projekt realisiert ist. Auch bei diesem Generator ergibt sich ein Teil der Snort Konfiguration der für jedes generierte System identisch ist und somit in der Runtime definiert wird. Teil dieser gleichbleibenden Konfiguration ist die in Listing 6.6 dargestellte Regel. Diese

Snort

```
alert ip any any -> any any (
  ip_proto:!tcp;
  msg:"Non Application Traffic";
  classtype:bad-unknown;
  sid:50000000;
  rev:1;
)
```

Listing 6.6: Snort Regel, die alle Protokolle außer TCP als Alarm meldet.

veranlasst Snort alle Protokolle außer TCP als Alarm zu protokollieren. Anhand dieser Regel sieht man den grundlegenden Aufbau einer Snort Regel, die mit dem Schlüsselwort `alert` beginnt, um zu kennzeichnen, dass bei einer Verletzung der Regel ein Alarm gemeldet wird. Als zweiter Parameter folgt die Angabe des Protokolls auf das sich die Regel bezieht gefolgt von zwei Parametern die die Quell-IP-Adresse und bei TCP Regeln den TCP Quellport angeben. Nach dem `->` werden Zieladresse und -Port angegeben gefolgt von weiteren Regeloptionen in einer Klammer, die in beliebiger Reihenfolge angegeben werden können. Mit `msg` wird hier die Meldung definiert, die im Falle eines Alarms ausgegeben wird und `sid` sowie `rev` werden verwendet um eine Regel und deren Version zu identifizieren. Diese Versionierung von Regeln wird verwendet, um Änderungen nachzuverfolgen und Aktualisierungen effizient durchführen zu können. In Listing 6.6 wird die Option `ip_proto:!tcp` verwendet, um alle Pakete die nicht zu TCP gehören zu erkennen. Dabei wird das Ausrufezeichen in Snort Regeln als Negationssymbol verwendet, um den darauf folgenden Wert zu negieren.

Um TCP Kommunikation zu erkennen, die im MontiSecArc-Modell nicht vorgesehen ist wird für jeden Konnektor eine Snort Regel generiert, die nur TCP Verbindungen von der IP Adresse des Clients zum Port des Servers der im Modell vorgesehen ist zulässt.

FM

```
<#setting number_format="##0">
<#assign c = 50000000>

<#list op.getValue("rules") as rule>
<#assign c = c+1>
alert tcp ! [${rule.getSrc()}] any -> [${rule.getDst()}]
      [${rule.getPorts()}] (
  msg:"Non Application Traffic";
  flags: S;sid:${c};
  rev:${rule.getRev()}
)
</#list>
```

Listing 6.7: Freemarker Template das die Regeln für Snort generiert.

Das Freemarker Template in Listing 6.7 verwendet die aus dem Modell berechneten Regeln, die in der Variable `rules` durch den `op` bereitgestellt werden und bereits wie in Abschnitt 6.2 beschrieben von Komponenten und Ports auf IP Adressen und TCP Ports übersetzt wurden. Das

Regel Objekt wird durch das Template im wesentlichen in die Snort Regel Syntax übersetzt. Dabei wird die Negation verwendet um die Quelladresse der Regel zu invertieren, da durch das MontiSecArc-Modell genau diese Adresse als gültige Absenderadresse vorgesehen ist.

Kapitel 7.

Zustandsbasierte Security-Tests für Webapplikationen

Kommunikationsprotokolle zur Informationsübertragung eignen sich gut zur Blackbox Wiederverwendung, da sie eine klare Schnittstelle bereitstellen. Werden zusätzliche Funktionen wie Sessions, Verschlüsselung oder Authentication als optionale Erweiterungen hinzugefügt, wie beispielsweise bei HTTP muss der Entwickler diese explizit aktivieren, um sie zu nutzen. Wie immer bei Security-Schutzmaßnahmen ist es notwendig, dass diese bei jeder Verwendung des Protokolls aktiviert werden, sodass keine Vulnerability entsteht.

In diesem Kapitel werden zunächst typische in der Literatur beschriebenen Arten von Vulnerabilities im Session Management von Webapplikationen beschrieben. Im Anschluss wird der in dieser Arbeit untersuchte Ansatz vorgestellt, der dem Prinzip des modellgetriebenen Security-Testing [SGS12] folgt, um diese Vulnerabilities zu identifizieren. Dabei werden UML/P Statecharts [Rum11, Sch12] als funktionales Verhaltensmodell der Anwendung verwendet, um automatisiert Security-Tests gegen die Anwendung auszuführen. Im Kontext der in Kapitel 4 vorgestellten Security Architektur Modellierung fokussiert dieser Ansatz auf die Interaktion von Komponenten. Diese Interaktion wird in MontiSecArc vereinfacht als struktureller `connector` dargestellt, wodurch die im Folgenden beschriebenen Angriffe erst in der hier vorgestellten Modellierung des Verhaltens identifiziert werden können.

7.1. Angriffe auf das HTTP Session Management

Es gibt verschiedene in der Literatur bekannte Angriffe auf das Session Management von Webapplikationen, die im folgenden dargestellt werden. Diese Angriffe zielen nicht darauf ab das Verfahren zur Authentication eines Nutzers zu beeinflussen, sondern dieses zu umgehen, indem die Session, welche nach einer erfolgreichen Authentication etabliert wird, gefälscht wird. Für das Session Management, also das Erstellen und Terminieren, sowie das Aufrechterhalten einer Session über mehrere HTTP Anfragen hinweg, gibt es verschiedene Implementierungsvarianten. Hier wird die Implementierung mit Cookies betrachtet, welche einen Nutzer über mehrere Anfragen hinweg identifizieren, wie dies im HTTP Standard vorgesehen ist [Bar11]. Andere Implementierungen beispielsweise mit versteckten Formularfeldern sind möglich, bieten jedoch weniger Schutz gegen Cross-Site-Scripting und sind daher nicht empfehlenswert. Damit ein Cookie als Session-ID und Authentication-Merkmal verwendet werden kann, setzt der Server den Cookie-Wert in seiner Antwort für einen Client auf einen langen Zufallswert, sodass ein Angreifer nicht durch bloßes raten eine gültige Session-ID erhalten kann.

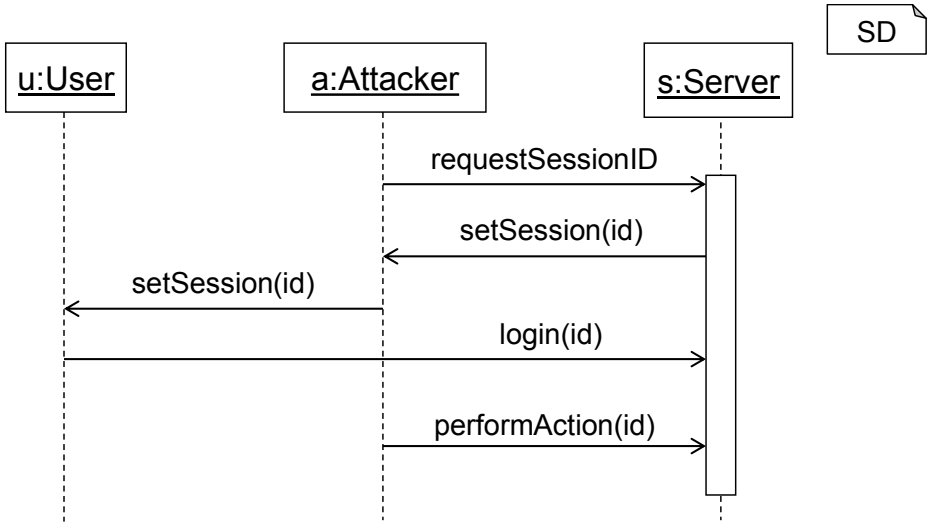


Abbildung 7.1.: Ablauf eines Session Fixation Angriffs.

Angriffe auf das HTTP Session Management werden nicht direkt von einem Angreifer durchgeführt, sondern dieser instrumentiert den Webbrowser eines Opfers diese Angriffe auszuführen. Der Browser ist somit ein Confused Deputy [Har88], der zwei Herren dient: Er stellt Anfragen an Webseiten und führt dazu die Authentication für den Nutzer durch. Gleichzeitig stellt der Browser Webseiten dar, die von einem Angreifer präpariert werden. Die Webapplikation kann somit Anfragen von Nutzern nicht von denen durch den Angreifer ausgelösten Anfragen unterscheiden.

Session Fixation

Definition 7.1.1 (Session Fixation Angriff) *Ein Session Fixation Angriff [Kol02] wird möglich, wenn die folgenden zwei Vulnerabilities in einer Webapplikationen zusammen auftreten:*

Die von der Anwendung vergebene Session-ID lässt sich durch einen Angreifer kontrollieren, bevor die Authentication eines Nutzers durchgeführt wird.

Nach der Authentication des Nutzers wird die vergebene Session-ID nicht invalidiert, sondern weiter verwendet. [Kol02]

Oft ist es möglich die Session-ID zusätzlich als GET Parameter in der URL zu setzen, sodass die Webapplikation diese übernimmt und in der Antwort als Cookie setzt. Somit muss ein Angreifer lediglich ein Element in einer beliebigen Webseite, die sein Opfer lädt, kontrollieren und die URL dieses Elements auf die verwundbare Webapplikation umleiten, um die Session bei seinem Opfer zu setzen.

Durch diese Voraussetzungen kann ein Angreifer den in Abbildung 7.1 dargestellten Angriff durchführen, bei dem er zunächst eine gültige Session-ID von der Webapplikation bezieht und diese verwendet, um die Session-ID des Nutzers auf einen ihm bekannten Wert zu fixieren. Logt sich der Nutzer nun ein, kann der Angreifer die ihm bekannte Session-ID verwenden, um die Webapplikation im Namen des Nutzers zu benutzen.

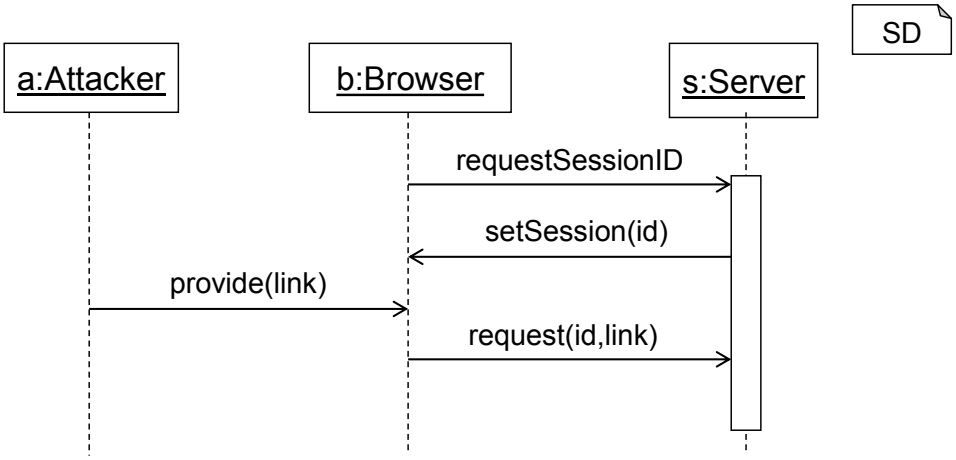


Abbildung 7.2.: Ablauf eines Cross-Site Request Forgery Angriffs.

Da für eine erfolgreiche Durchführung des Angriffs beide Vulnerabilities notwendig sind, reicht es aus eine der Beiden zu schließen. Für eine robuste Vermeidung dieses Angriffs ist es allerdings sinnvoll beide Vulnerabilities zu schließen.

Cross-Site Request Forgery

Definition 7.1.2 (Cross-Site Request Forgery (CSRF)) *Bei einer Cross-Site Request Forgery Vulnerability führt einer Webanwendung auf eine GET oder POST Anfrage eines Browsers hin eine Aktion aus ohnezu prüfen, ob diese Anfrage aus einem Formular stammt, welches die Webapplikation zuvor an einen Nutzer ausgeliefert hat. [Wat01]*

CSRF wird auch Session Riding [Sch04] oder XSRF [Bur05] genannt. Analog zum Cross-Site-Scripting gibt es die beiden Varianten Reflected CSRF und Stored CSRF.

Diese Vulnerability nutzt ein Angreifer wie in Abbildung 7.2 dargestellt aus, indem er ein präpariertes HTML Dokument an den Browser eines Nutzers schickt, der beim parsen des Dokuments einen vom Angreifer bestimmten Link im Kontext des Nutzers aufruft. Beim Reflected CSRF schickt der Angreifer dazu dem Nutzer eine Nachricht z.B. via Email in der HTML enthalten ist oder ein Link, den der Nutzer anklickt oder verpackt diesen Link in einen QR-Code, den der Nutzer einscannt. Alternativ kann der Angreifer den HTML Code auch in einer Webseite abseits der Webanwendung unterbringen, die Nutzer häufig besuchen. Bietet die Webanwendung selber dem Angreifer gewollt oder ungewollt die Möglichkeit HTML zu speichern, kann der Angreifer den präparierten HTML Code dort ablegen, was als Stored CSRF bezeichnet wird. Muss ein Nutzer der Webapplikation angemeldet sein, bevor er auf das vom Angreifer platzierte HTML zugreifen kann, hat ein solcher Angriff eine hohe Erfolgswahrscheinlichkeit.

Ganz gleich auf welchem Weg der vom Angreifer kontrollierte HTML Code in den Browser gelangt, wird der Browser diesen zunächst parsen. Nach dem Parsen lädt der Browser Elemente wie Bilder, deren Quelle im HTML als URL definiert wurden, durch eine GET Anfrage an die definierte URL nach. Um eine Aktion in der Webapplikation im Namen des Nutzers durchzuführen,

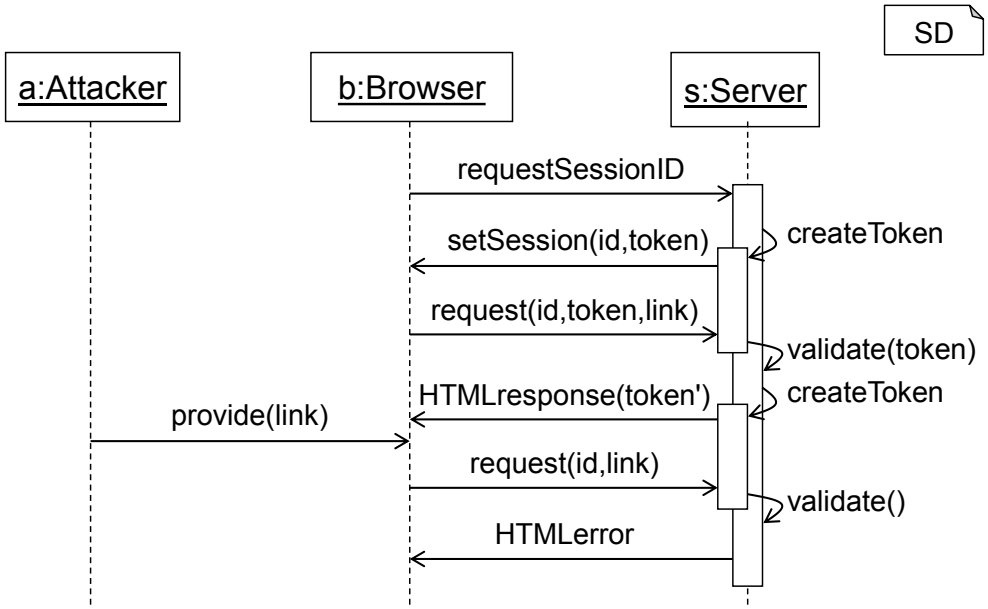


Abbildung 7.3.: Ablauf einer Anfrage mit CSRF-Token als Schutz gegen CSRF.

die mit einer GET Anfrage an eine bestimmte URL ausgelöst wird, setzt der Angreifer diese URL als Quelle eines Bildes, sodass der Browser beim Nachladen des Bildes die Aktion ausführt.

Für eine Aktion, die durch eine POST Anfrage ausgelöst wird, verwendet der Angreifer statt eines Bildes ein Formular, welches er mittels JavaScript automatisch nach dem Parsen abschickt. Da für diesen Angriff JavaScript notwendig ist, wird er häufig mit Cross-Site-Scripting verwechselt. Beim Reflected CSRF wird das JavaScript allerdings nicht in die Webanwendung eingeschleust, sondern ganz legal im Kontext einer anderen, durch den Angreifer kontrollierten Webseite ausgeführt. Über Stored CSRF eine POST Aktion auszuführen macht wenig Sinn, da dies tatsächlich eine Cross-Site-Scripting Vulnerability in der Webanwendung voraussetzt, die es einem Angreifer bereits erlaubt sämtliche Aktionen des Nutzers in der Applikation durchzuführen.

Als Schutzmaßnahme gegen Session Fixation in jeder Antwort des Servers einen neuen Cookie zu vergeben beschränkt die möglichen Aktionen, die ein CSRF Angriff ausführen kann auf solche, die von der zuletzt angefragten Seite aus erlaubt sind. Dies würde die Auswirkungen eines CSRF Angriffs einschränken, ihn jedoch nicht verhindern. Um den Angriff vollständig zu verhindern, ist es notwendig, wie in Abbildung 7.3 dargestellt, in jede Antwort des Servers ein CSRF-Token einzubauen, welches bei allen Anfragen, die der Browser aus dieser Webseite heraus stellt, mit an den Server übertragen wird. Als CSRF-Token wählt der Server entweder einen für jede Antwort zufälligen Wert und überprüft vor der Durchführung einer Aktion, ob dieser Wert in der Anfrage des Browsers enthalten war, oder es wird mittels einer kryptographischen Hashfunktion die Echtheit des CSRF-Token sichergestellt. Im beispielhaften Ablauf in Abbildung 7.3 loggt sich der Benutzer zunächst ein, sodass er eine Session-ID und ein CSRF-Token erhält. Die Session-ID sendet der Browser bei jeder folgenden Anfrage, das CSRF-Token nur bei der unmittelbar

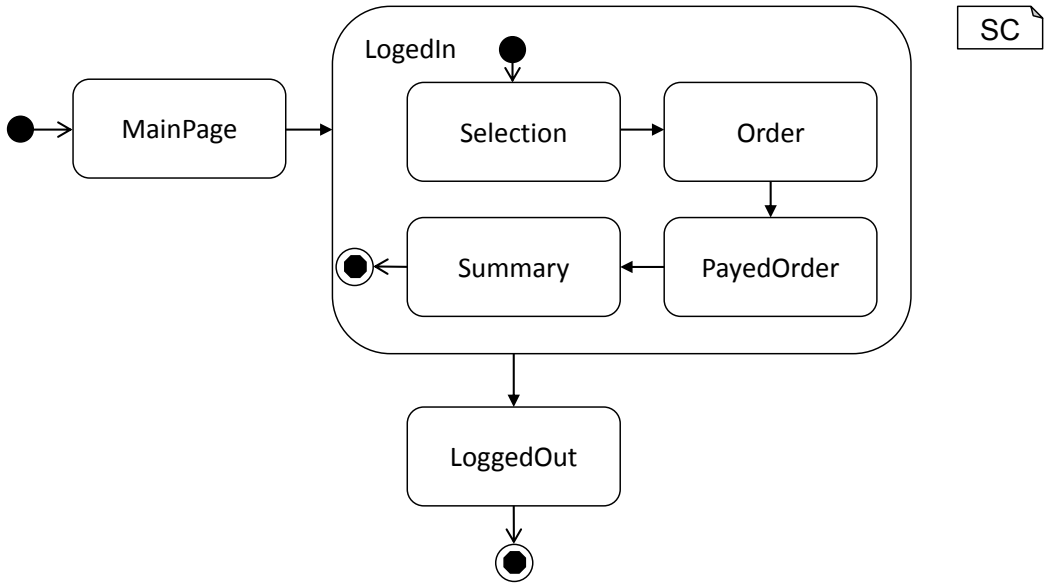


Abbildung 7.4.: Das Statechart beschreibt das Verhalten einer Webseite zur Pizzabestellung.

Nächsten. Der Server überprüft jeweils die Gültigkeit eines erhaltenen CSRF-Token und antwortet auf Anfragen mit ungültigem CSRF-Token durch eine Fehlerseite ohne die Anfrage auszuführen. Somit ist es dem Angreifer nicht möglich Aktionen in der Webapplikation durchzuführen, da der Link, den er dem Browser seines Opfers unterschiebt, keinen CSRF-Token enthält.

Business Logic Error

Neben den zuvor beschriebenen Angriffen, die darauf abzielen die Authentication zu umgehen, kann ein Angreifer, der einzelne in einer Webanwendung vorgesehene Aktionen überspringen kann, ebenfalls einen erheblichen Schaden für den Betreiber der Webanwendung verursachen. Das Statechart in Abbildung 7.4 beschreibt den vom Entwickler beabsichtigten Ablauf einer Pizzabestellung in einer Webanwendung. Dabei wählt der Nutzer nach dem Login eine Pizza aus, anschließend bezahlt er die Pizza und zuletzt wird eine Zusammenfassung der Bestellung angezeigt. Überprüft diese Applikation nicht, dass vor dem Backen der Pizza die Bezahlung durchgeführt wurde oder verlässt sie sich darauf, dass der Browser diese Information mitliefert kann ein Angreifer die Bezahlung überspringen, indem er direkt die Anfrage sendet, die ihn in den Zustand Backen bringt.

Da HTTP grundsätzlich nicht darauf ausgerichtet ist eine Reihenfolge zwischen Anfragen sicherzustellen, muss der Entwickler dies in der Anwendungslogik explizit prüfen und kann sich nicht darauf verlassen, dass die Webseiten nur in der Reihenfolge aufgerufen werden, in der sie dem Nutzer präsentiert werden. Hier gibt es viele Möglichkeiten dies zu realisieren, die alle auf einem sicheren Session Management aufsetzen.

7.2. Identifikation von Session Management Vulnerabilities

Ein guter Ansatz Session Management Vulnerabilities zu vermeiden ist natürlich das Session Management mit bestehenden Bibliotheken und Frameworks zu realisieren, die keine bekannten Vulnerabilities besitzen. Damit ist jedoch nicht gegeben, dass Entwickler alle möglichen Abläufe, die in einer Webapplikation möglich sind, überblicken. Daher bietet sich ein modellgetriebener Ansatz an, bei dem Modelle wie Aktivitätsdiagramme [RR13] oder Statecharts [BKK11] verwendet werden, um den Ablauf in einer Webapplikation durch das Modell zu definieren.

Zur Überprüfung einer gegebenen Applikation auf bekannte Vulnerabilities im Session Management existieren Testwerkzeuge wie die Burp Suite [Por15], Zed Attack Proxy [Ope15d] oder Accunetix Web Scanner [Acu15], die einen Blackbox Ansatz verwenden und versuchen aus den Antworten einer Webapplikation die Session-ID und das CSRF-Token zu extrahieren. Liefert die Webanwendung auf manipulierte Anfragen, die kein CSRF-Token enthalten eine Antwort, die einer Antwort mit CSRF-Token ähnelt, so wird eine CSRF Vulnerability gemeldet. Da durch den Blackbox Ansatz nur ein beschränktes Wissen über die Applikation und insbesondere ihre Logik vorliegt, können diese Tests kein abweichendes Verhalten der Applikation von der intendierten Business Logik identifizieren. Ein erster Ansatz der jede Abweichung im Antwortverhalten einer Webapplikation von dem in einem Statechart spezifizierten Verhalten als Fehler meldet wird von Busch et al. [BOS13] beschrieben.

7.2.1. Funktionales Session-Modell

SC

```
1  statechart PizzaShop {
2    initial state MainPage {
3      entry / {
4        url("http://test/pizza/");
5        assertTitleMatch("Welcome");
6      }
7    }
8    final state LoggedOut {
9      entry / { assertImagePresent("partner.jpg","partners"); }
10   }
11
12   MainPage -> LoggedIn : / {
13     setTextField("name", "testUser");
14     setTextField("pass", "word");
15     submit();
16   }
17   LoggedIn -> LoggedOut : / { clickLink("logout"); }
```

Listing 7.5: Dieser Teil des UML/P Statecharts beschreibt den öffentlichen Bereich einer Webapplikation zur Pizzabestellung.

Im Folgenden wird ein Ansatz vorgestellt, der dem modellgetrieben Security-Testing Prinzip folgt, und dementsprechend ein funktionales Modell der zu testenden Webapplikation verwen-

```
18   <<privilege="1">> state LoggedIn {                                SC
19     entry / {
20       setIdleTimeout("3600");
21       assertLinkPresent("logout");
22     }
23
24     initial state Selection {
25       entry / {assertTitleMatch("Select Pizza");}
26     }
27     state Order {
28       entry / {
29         assertTitleMatch("Your Order");
30         assertFormPresent("credit-card");
31       }
32     }
33     <<privilege="2">> state PayedOrder {
34       entry / {
35         assertTitleMatch("Your Order");
36         assertFormNotPresent("credit-card");
37       }
38     }
39     final state Summary {
40       entry / {assertTitleMatch("Thank You");}
41     }
42
43     Selection -> Order : / { clickLink("pizza-salami"); }
44     Order -> PayedOrder : / {
45       setTextField("card", "4363291446185929"); submit();
46     }
47     PayedOrder -> Summary : / { clickLinkWithExactText("Order"); }
48   }
49 }
```

Listing 7.6: Der LoggedIn Zustand der Webapplikation zur Pizzabestellung wird in diesem Teil des UML/P Statechart dargestellt.

det, um Security-Tests durchzuführen, die speziell auf diese Anwendung zugeschnitten sind. Dieses Prinzip erlaubt es Entwicklern Security-Tests durchzuführen, ohne bei der Modellierung Security-Wissen einzubringen, so wie es Anforderung A3 fordert. Als funktionales Modell wird ein UML/P Statechart verwendet, welches mit jedem Zustand eine Webseite beschreibt, die von der Webapplikation an den Browser ausgeliefert wird und dessen Transitionen die vom Entwickler vorgesehenen möglichen Übergänge zwischen diesen Webseiten definieren. Die Listings 7.5 und 7.6 zeigen eine Erweiterung des zuvor beschriebenen Beispiels aus Abbildung 7.4 in textueller UML/P Statechart Notation, bei der sowohl technische als auch inhaltliche Erweiterungen dargestellt sind, die im Folgenden beschrieben werden.

Um die Zustände, also die Webseiten, genauer zu beschreiben, wird für jeden Zustand ange-

geben, wie die Webseite aussehen muss, damit das Statechart einen Zustand annimmt. Diese Vorbedingung wird in der entry-Aktion eines Zustands in Java beschrieben (Zeile 5). Dabei wird das JWebUnit Framework [JWe15] verwendet, um die Bedingungen zu formulieren und nur wenn die JWebUnit Tests keinen Fehler melden, ist die Vorbedingung erfüllt und der Zustand wird angenommen. In der entry-Aktion kann durch die Methode `url` die URL angegeben werden, unter der die Webseite erreichbar ist. Im initialen Zustand `MainPage` wird dies in Zeile 4 genutzt, um die Startseite der Webapplikation festzulegen.

Durch die Transitionen wird beschrieben, zwischen welchen Zuständen ein Zustandsübergang möglich ist und durch die Aktionen, die für eine Transition angegeben werden, wird die HTTP Anfrage an die Webapplikation definiert, durch die die Transition durchgeführt wird. Hier kommt wieder JWebUnit zum Einsatz, um diese Anfrage zu beschreiben. So wird die Transition von `Selection` zu `Order` in Zeile 43 durch einen Klick auf den Link mit der ID `pizza-salami` durchgeführt.

Um Session Fixation testen zu können werden im funktionalen Modell die Zustände markiert, für die ein Nutzer spezielle Berechtigungen benötigt. Hier wird dazu der Stereotyp `privilege` verwendet, der mit einer Zahl das Berechtigungsniveau angibt. Im Pizza Shop wird der hierarchische Zustand `LoggedIn` mit dem Berechtigungsniveau 1 versehen, wodurch sämtliche Subzustände dieses Berechtigungsniveau erhalten, bis explizit ein anderes Berechtigungsniveau angegeben wird, wie im Zustand `PayedOrder` der das Berechtigungsniveau 2 hat.

Für einen Zustand kann in der entry-Aktion durch die Methode `setIdleTimeout` angegeben werden nach wie vielen Sekunden die Session ungültig wird. Diese Information wird zum Testen des Session Timeout verwendet und in Zeile 20 in Listing 7.6 auf eine Stunde gesetzt.

7.2.2. Security-Tests

Auf Basis des zuvor beschriebenen funktionalen Modells der Webapplikation werden die im OWASP Testing Guide [Ope15c] aufgelisteten Tests für Session Management und Business Logik automatisiert. Dabei wird ein generativer Ansatz verfolgt, bei dem die im funktionalen Modell beschriebenen Informationen genutzt werden, um die zu testende Webapplikation zu steuern. Im Folgenden wird zunächst die Übersetzung des funktionalen Modells in diese Steuerung und die Implementierung des dazugehörigen Generators beschrieben. Anschließend werden Security-Tests dargestellt, die diese Steuerung verwenden, um Sessions der Webapplikation in die für den Test notwendigen Zustände zu bringen.

Implementierung des Generators

Wie Abbildung 7.11 darstellt verwenden die Security-Tests die Interfaces `State` und `Transition` zur Steuerung der Webapplikation. Listing 7.7 und 7.8 zeigen die Methoden, welche den Security-Tests dabei zur Verfügung stehen. Für jeden Zustand aus dem zuvor beschriebenen funktionalen Modell wird eine Java Klasse generiert, die das Interface `State` implementiert. Somit können Attribute der Zustände aus dem Modell wie deren URL oder Session Timeout in Tests verwendet werden. Über die Methode `assertState` kann ein Test sicherstellen, dass sich die Session der Webapplikation in einem bestimmten Zustand befindet. Die aus dem Zustand `Order` generierte Methode `assertState` ist in Listing 7.9 dargestellt. Diese beinhaltet die

```java
package de.rwth.se.pizzashop.interfaces;

import java.util.ArrayList;

public interface State {
  public String getName();
  public int getPrivilege();
  public String getStateProperty();

  public ArrayList <Transition> getTransitions();
  public String getUrl();
  public void setUrl(String url);
  public int getSessiontime();

  public void assertState();
}
```

MSA

Listing 7.7: Java Interface welches die Informationen über einen Zustand im funktionalen Modell bereitstellt.

```java
package de.rwth.se.pizzashop.interfaces;

public interface Transition {
   public String getName ();
   public State getSourceState();
   public State getTargetState();
   public void executeTransition();
   public boolean ignoreXSRF();
}
```

MSA

Listing 7.8: Dieses Java Interface stellt die Informationen über eine Transition im funktionalen Modell bereit.

in der entry-Aktion des Zustands definierten JWebUnit Tests. Da JWebUnit Tests im Fehlerfall über eine Exception beendet werden, die im Testframework behandelt wird, besitzt die Methode `assertState` keinen Rückgabewert und alle Tests die nach dieser Methode ausgeführt werden können sich darauf verlassen, dass sich die Session in dem durch die entry-Aktion definierten Zustand befindet.

Um die Webapplikation zu steuern werden die im funktionalen Modell definierten Transitionen verwendet. Dazu wird jede Transition des Modells in eine Klasse übersetzt, welche durch den Quell- und Zielzustand benannt wird. So zeigt Listing 7.10 die Klasse `MainPageToLoggedIn`, welche aus der Transition vom Zustand `MainPage` zum Zustand `LoggedIn` resultiert. Die dargestellte Methode `executeTransition` führt die im Modell spezifizierte Transition mit Hilfe von JWebUnit durch. Diese Methode der entsprechenden Klassen wird von den im Folgenden beschriebenen Tests genutzt, um gezielt einzelne Transitionen in der Session der Webanwendung

```
1 package de.rwth.se.pizzashop.states;                    MSA
2
3 import de.rwth.se.pizzashop.interfaces.State;
4 import static net.sourceforge.jwebunit.junit.JWebUnit.*;
5
6 public class Order implements State {
7
8   public void assertState() {
9     assertTitleMatch("Your Order");
10    assertFormPresent("credit-card");
11  }
12  ...
13 }
```

Listing 7.9: Die `assertState` Methode der generierten Klasse Order.

```
1 package de.rwth.se.pizzashop.transitions;                MSA
2
3 import de.rwth.se.pizzashop.interfaces.Transition;
4 import static net.sourceforge.jwebunit.junit.JWebUnit.*;
5
6 public class MainPageToLoggedIn implements Transition {
7
8   public  void executeTransition() {
9     setTextField("name", "testUser");
10    setTextField("pass", "word");
11    submit();
12  }
13  ...
14 }
```

Listing 7.10: Die `executeTransition` Methode der generierten Klasse MainPageToLoggedIn.

durchzuführen. Durch die eingesetzten Interfaces können gleichzeitig die Tests unabhängig von der konkreten Webanwendung formuliert werden. Die durch die Security-Tests direkt genutzten Funktionalitäten von JWebUnit sind unabhängig von der zu testenden Anwendung.

CSRF, Business Logik und Logout Test

In einem CSRF Test wird überprüft, ob Transitionen auch durchgeführt werden können, wenn die Session sich nicht im dafür vorgesehenen Zustand befindet. Im Hinblick auf die im Statechart modellierte Business Logik lässt sich eine solche Situation auch als Transition verstehen, die nicht vorgesehen war oder nicht modelliert wurde und die es einem Angreifer unter Umständen erlaubt das Verhalten der Webapplikation zu seinen Gunsten zu verändern.

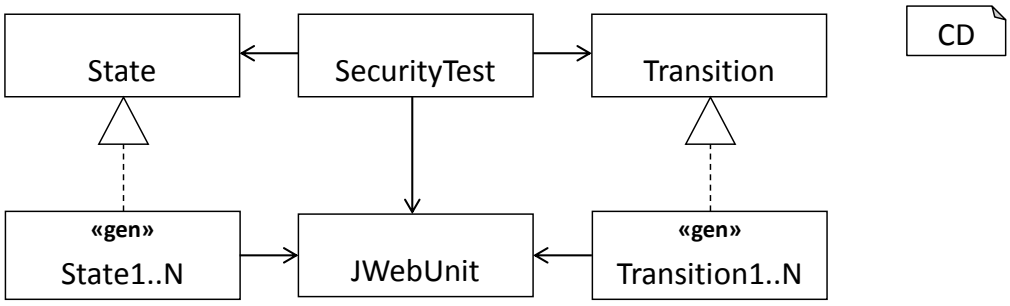


Abbildung 7.11.: Struktureller Aufbau der Security-Tests.

Um das Verhalten der Webapplikation bei einem solchen Angriff zu testen, müssen die durch JWebUnit Code beschriebenen Aktionen einer Transition ausgeführt werden, ohne dass zuvor die Webseite, auf der diese Aktion beschrieben wird geladen wird. Da JWebUnit auf den HTML Elementen einer geladenen Webseite arbeitet, ruft die Methode `clickLink` unter Umständen zuerst eine JavaScript Methode auf, die für den entsprechenden Link definiert ist, bevor eine Anfrage an den Server gesendet wird. Daher lassen sich die Anfragen an die Webapplikation nicht direkt aus dem JWebUnit Code ableiten, sondern nur bei Ausführung von validen Transitionen beobachten. Zur Vorbereitung des CSRF Tests werden daher alle Transitionen des Statecharts mindestens einmal durchgeführt und dabei die ausgeführte HTTP Anfrage zur späteren Ausführung aufgezeichnet. Dabei werden Cookies und andere HTTP Header nicht gespeichert, da diese durch den Zustand der Session während des Security-Tests bestimmt werden.

Da eine Webapplikation je nach Zustand der Session, beispielsweise ob ein Nutzer eingeloggt ist oder nicht, anders auf Anfragen reagiert, werden die aufgezeichneten Anfragen von jedem Zustand der Webapplikation aus ausgeführt. Ob eine solche Anfrage erfolgreich eine neue Transition offenbart hat, die im Statechart nicht vorgesehen war, wird festgestellt indem die Antwort der Webapplikation mit der Vorbedingung des Zielzustands der Transition überprüft wird. Ist die Vorbedingung erfüllt, wurde der Zielzustand angenommen und eine neue Transition gefunden, die eine CSRF bzw. Business Logik Vulnerability darstellt.

Eine spezielle Instanz dieser Vulnerability entsteht, wenn die Logout Funktion des Session Managements nicht funktioniert. Dann ist es möglich Transitionen auszuführen, für die eine Authentication notwendig ist, obwohl die Session in einem ausgeloggten Zustand ist.

Dieser Ansatz kommt ohne explizite Modellierung oder heuristische Erkennung eines CSRF-Token aus, sogar der Stereotyp `privilege` ist für diesen Test nicht notwendig. Fehler in der Implementierung des CSRF-Token, beispielsweise dass dieses nicht zufällig ist oder eine beliebige Transitionen mit einem aktuellen CSRF-Token genau einmal ausgeführt werden kann, werden von diesem Test nicht gefunden.

Session Fixation und Cookie Test

Eine Session Fixation Vulnerability liegt vor, wenn nach einer Authentication, also der Erhöhung des Berechtigungsniveaus, kein neuer Cookie durch die Webapplikation gesetzt wird. Um eine Webapplikation auf diese Vulnerability zu testen, werden daher alle Transitionen durchgeführt, deren Ausgangszustand ein geringeres Berechtigungsniveau als ihr Ziel haben. Dabei werden die Cookies vor und nach der Transition verglichen und wenn diese identisch sind, wird eine Session Fixation Weakness gemeldet. Wie bereits zuvor erwähnt, kann eine durch den Test identifizierte Weakness nicht unbedingt ausgenutzt werden, da hierzu die Webapplikation das explizite setzen der Session-ID erlauben muss.

Bei dieser Analyse der Cookies wird außerdem geprüft, ob das `secure` und `HttpOnly`-Flag gesetzt ist, welches verhindert, dass der Cookie in einem XSS-Angriff ausgelesen werden kann. Außerdem wird aus den für den CSRF Test gespeicherten Anfragen die gemeinsame Domain aller Anfragen errechnet. Das Domain Attribut sollte mindestens auf diese Domain oder eine Subdomain gesetzt werden. Analog wird ein maximal gemeinsamer Wert für das Path Attribut ermittelt und überprüft, ob dieses gesetzt ist.

Session Timeout und Denial of Service Test

Den Session Timeout einer Webapplikation zu testen ist sehr zeitintensiv und ein Denial of Service (DoS) Test sehr ressourcenintensiv. Daher sollten diese Tests nur im nightly Build durchgeführt werden, bei dem die Ausführungszeit der Tests keine Rolle spielt.

Um zu prüfen, ob Session Cookies nach einer gewissen Zeit ablaufen und nicht mehr gültig sind, wird im Session Timeout Test die Session in einen Zustand versetzt, der mit `setIdleTimeout` angibt, wie lange eine Session hier gültig sein soll. Dementsprechend wartet der Test die angegebene Zeit und führt danach eine Transition zu einem Zustand mit gleichem Idel-Timeout aus. Mittels der Vorbedingung des Zielzustands wird geprüft, ob die Transition erfolgreich war und somit eine Session Timeout Vulnerability existiert.

Um eine DoS Situation des Session Managements zu provozieren und zu testen, ob eine Applikation diesem Angriff standhält, werden aus dem funktionalen Modell mögliche Pfade zu allen Zuständen berechnet und jeder dieser Pfade wird von einer zu definierenden Menge von Threads bearbeitet. Die Threads führen kontinuierlich die in den Transitionen ihres Pfades definierten Aktionen in der Webapplikation aus und wenn ein Thread am Ziel seines Pfads ankommt, beginnt er erneut am Startzustand. Somit werden für jeden Zustand viele Sessions erzeugt, die genau in diesem Zustand sind. Gibt es nun einen oder mehrere Zustände, die besonders viel Ressourcen auf dem Server verbrauchen und der Server gibt keine Ressourcen von alten Sessions frei, entsteht ein DoS, welcher sich durch erhöhte Antwortzeiten bzw. eine nicht antwortende Webapplikation beobachten lässt.

Kapitel 8.

Sichere Eingabe- und Ausgabeverarbeitung mit MontiCoder

> A web application is a transducer from a broken thing to a broken thing.
>
> Sergey Bratus auf dem zweiten LangSec IEEE S&P Workshop

In diesem Kapitel werden zunächst die Probleme bei der Verarbeitung von Eingabedaten erläutert, die zu einem Großteil der Security Bugs in Software führen. Darauf folgt eine Übersicht über Ansätze, die bereits Teile dieser Probleme lösen. Im Anschluss wird eine genaue Definition des Problems des korrekten Unparsens angegeben und danach ein Ansatz vorgestellt, der darauf abzielt die Entwicklung und Verwendung von kontextsensitiven Encodern für beliebige Sprachen und selbst entwickelte Formate zu vereinfachen. Dazu wird die Definition des Encodings explizit in die kontextfreie Grammatik integriert, sodass Unparser, kontextsensitive Encoder, Parser und kontextsensitive Decoder daraus mithilfe von MontiCoder, einer Erweiterung von MontiCore, generiert werden können. Diese generierten Werkzeuge zur Verarbeitung von Ein- und Ausgaben stellen sicher, dass Eingabedaten weder die Struktur der Ausgabe, noch deren vom Entwickler vorgesehene Semantik verändern. Durch die Nutzung dieser generierten Werkzeuge werden Entwickler von der Aufgabe des für Security relevanten Encoding befreit (Anforderung A3 und A4) und Injection Vulnerabilities verhindert.

Durch die Integration des Encodings in die Sprachdefinition wird das Problem des korrekten Encodens von den Schultern der Entwickler, die eine Sprache nutzen auf jene übertragen, die die Sprache entwickeln. Letztere sollten sich besser mit den Spezialfällen der Sprache auskennen, die sie selbst eingebaut haben.

Ein inhärentes Problem von Systemen die Eingaben verarbeiten, welches als Confused Deputy [Har88] bezeichnet wird, lässt sich nicht alleine durch die Security-Architektur lösen sondern erfordert spezielle Schutzmaßnahmen bei der Datenverarbeitung im Programmcode. Das Confused Deputy Problem tritt auf, wenn ein System oder Prozess mit hohen Privilegien Daten

verarbeitet, die von einem Nutzer oder Angreifer mit niedrigen Privilegien erstellt werden. Dabei verwendet der Prozess die Eingabedaten, um eine vom Entwickler vorgesehene Funktion zu erfüllen, für die der Prozess höhere Privilegien benötigt. Ziel des Angreifers ist es die höheren Privilegien des Prozesses zu missbrauchen und mehr als die vorgesehene Funktion auszuführen. Je nachdem wie und zu welchem Zweck die Eingabedaten verarbeitet werden und welche Annahmen Entwickler über die Eingabedaten machen, die im Programm nicht geprüft werden, kann ein Angreifer unterschiedlich viele zusätzliche Funktionen ausführen. Diese zusätzlichen, vom Entwickler nicht vorgesehenen Funktionen bilden eine eigene Klasse von Bugs bei der Verarbeitung von Eingabedaten, die sogenannten Injection Vulnerabilities. Bekannte Injection Vulnerabilities werden unterschiedlich benannt, so werden bei SQL Injection Befehle in einen SQL-Befehl eingefügt und bei Cross-Site-Scripting wird JavaScript in HTML eingefügt.

Laut NVD sind mehr als 25% aller CVEs der vergangenen Jahre eindeutig auf Injection Vulnerabilities zurückzuführen. Allerdings werden in der NVD nur Software Produkte, die auch als solche erhältlich sind, gelistet. Also insbesondere keine Anwendungen, die speziell für ein Unternehmen entwickelt wurden und beispielsweise zur Überwachung und Steuerung der wichtigen Kernprozesse des Unternehmens eingesetzt werden. Meist besitzen solche Systeme eine Weboberfläche zur Nutzerinteraktion, für die die OWASP Top 10 [WO13], die häufigsten und kritischsten Vulnerabilities auflistet, um bei Entwicklern ein Bewusstsein für diese zu schaffen. In diesem Ranking belegen Injection Vulnerabilities den ersten Platz, wobei hier Cross-Site-Scripting (XSS), welches ebenfalls zu den Injection Vulnerabilities zählt, als eigenständige Vulnerability Platz drei belegt.

Eine XSS Vulnerability liegt vor, wenn eine Webapplikation Eingabedaten, die sie vom Browser beispielsweise als Parameter einer GET Anfrage erhalten hat, in der generierten HTML Antwort verwendet ohne zu überprüfen, ob die Eingabedaten HTML oder JavaScript enthalten. Ein Angreifer kann somit beliebigen JavaScript Code in die Seite einfügen und mit diesem die Webapplikation steuern. Durch diese Vulnerability kann ein Angreifer Aktionen in der Webapplikation im Namen eines Nutzers durchführen nachdem der Browser des Nutzers die GET Anfrage gestellt hat. Für einen erfolgreichen Angriff muss der Nutzer also eine Webseite aufrufen, die vom Angreifer kontrolliert wird oder eine beliebige unverschlüsselte Webseite aufrufen, wenn der Angreifer den Netzwerkverkehr beeinflussen kann.

Wie bereits die allgemeine Beschreibung des Confused Deputy Problems zeigt, sind Injection Vulnerabilities nicht auf Websysteme beschränkt, sondern können bei jedem Programm auftreten, dass Eingaben verarbeitet. Diese Ein- und Ausgaben eines Programms sind Dokumente[1], die zu der Ein- bzw. Ausgabesprache des Programms gehören. Diese Sprachen werden auch als Protokolle oder Formate bezeichnet und müssen sorgfältig entworfen werden, sodass Programme die selbe Sprache zur Kommunikation verwenden. Um diese Sprachen formal zu definieren sollten Kontextfreie Grammatiken verwendet werden, da diese in Parser übersetzt werden können. Denn nur mithilfe eines Parsers lässt sich in einem Programm erkennen, ob ein Dokument zu einer definierten Sprache gehört [BDL+14]. Diese formale Definition der Sprache und der Parser verhindert die Entstehung von Kommunikationsprotokollen, die unentscheidbar sind [SPBS11].

In der Praxis wird versucht solche unentscheidbaren Kommunikationsprotokolle durch ad-hoc Parser zu verarbeitet. Allerdings müssen diese aufgrund der Unentscheidbarkeit der zu

[1] Bezogen auf die modellgetriebene Softwareentwicklung könnte man diese Dokumente auch als Modelle verstehen.

erkennenden Sprache die Eingabedaten zumindest zum Teil ausführen, bevor entschieden werden kann, ob das zu parsende Dokument korrekt ist. Dies lässt sich ausnutzen, um Parser durch Eingabedaten so zu steuern, dass sie von einem Angreifer vorgegebene Befehle ausführt, sodass diese Parser zu Weird Machines werden [BDL+14]. Analog zum Empfangen und Parsen eines Dokuments wird beim Senden Unparsen eingesetzt, um die Datenstruktur, welche das zu sendende Dokument im Programm repräsentiert, in das Dokument auszugeben [ZB14] [2].

Grammatiken und (Un)Parser sind nicht die erste Wahl für Entwickler, die mit wenig Aufwand funktionale Anforderungen realisieren wollen. Daher müssen diese Konzepte in Werkzeuge integriert werden, um Anforderung A2 zu erfüllen und Entwicklern eine einfache Nutzung dieser formalen Konzepte zu ermöglichen. Nur so kann verhindert werden, dass scheinbar einfachere Verfahren wie Templates verwendet werden, bei denen alle Eingabedaten als String behandelt werden und ohne genauere Kenntnis der Ausgabesprache Variablen im Template mit diesen Daten besetzt und ausgegeben werden. Besonders verlockend ist die Verwendung solcher vereinfachter Verfahren bei menschenlesbaren Sprachen wie XML, HTML, JavaScript und Json. Diese Vereinfachung führt zu Programmen, die in den meisten Fällen gut funktionieren, solange sie keine wichtigen Aufgaben ausführen und Security keine wichtige Anforderung ist. Aus der Security-Perspektive sind die Vereinfachungen Vulnerabilities im (Un)Parser bei der Verarbeitung von Eingabedaten. Diese können ausgenutzt werden, um das Verhalten des Programms zu beeinflussen [BDL+14], oder im Fall des Unparsen das erstellte Dokument in einer vom Entwickler nicht erwarteten Weise zu beeinflussen.

Um Anforderung A3 und A4 zu erfüllen müssen Injection Vulnerabilities vollständig verhindert werden, sodass Entwickler sich nicht um diese kümmern müssen und sich auf die Business Logik konzentrieren können. Dazu müssen Unparser mit Encodern verwendet werden, die Eingabedaten bei der Ausgabe in ein Dokument encoden, sodass Schlüsselwörter und Kontrollzeichen der Ausgabesprache aus den Eingabedaten entfernt werden. Dies verhindert nachweislich (Anforderung A6), dass die von Entwicklern vorgesehene Semantik des Ausgabedokuments durch Eingabedaten nicht verändert wird. Gerade wegen seiner einfachen Lesbarkeit für Menschen ist HTML eine komplexe formale Sprache, die mehrere Kontexte besitzt, in denen Eingabedaten unterschiedlich encoded werden müssen. Eingabedaten können daher nicht im Voraus beispielsweise beim Parsen encoded werden, sondern müssen Kontext-spezifisch durch den Unparser behandelt werden. Der hier vorgestellte Ansatz befreit Entwickler bei der Generierung von Ausgabedokumenten von der manuellen, fehleranfälligen Anwendung von kontextspezifischen Encodern, indem bereits bei der Sprachdefinition Encodingregeln definiert werden, sodass Parser und Unparser generiert werden können, die dieses Encoding automatisch durchführen.

8.1. Bestehende Ansätze

Ansätze zur Vereinfachung der Verarbeitung von Formalen Sprachen haben eine lange Geschichte, die bei der Generierung von Parsern aus Grammatiken [Aho03] und der automatischen Formatierung von Ausgabedokumenten [Hug95, Wad98], dem Pretty-Printing beginnt. Pretty-Printing

[2] Unparsen wird auch als Serialisierung und Parsen als Deserialisierung bezeichnet.

konzentriert sich darauf einen AST oder Parsetree so in ein Textdokument auszugeben, dass dieses für Menschen möglichst gut lesbar ist. Allgemein wird die Übersetzung eines AST in ein Dokument, bei dem das Layout des Dokuments nicht unbedingt eine Rolle spielt als unparsing bezeichnet [ZB14]. Unparser sind spezifisch für eine Sprache, sodass Änderungen an der Grammatik und dem daraus generierten Parser ebenfalls Änderungsbedarf am Unparser verursachen. Um Inkonsistenzen zu vermeiden lassen sich Pretty-Printer aus Kontextfreien Grammatiken generieren [vdBV96] und auch die umgekehrte Richtung ist möglich [MW13]. Die Definition von Parser und Pretty-Printer zusammen ist auch als eingebettete DSL in einer Programmiersprache möglich, bei der die Funktionen invertierbar sind [RO10].

HTML ist besonders anfällig für Injection Vulnerabilities, da es viele unterschiedliche Kontexte besitzt, in denen unterschiedliche Encodings angewandt werden müssen. So muss beispielsweise das Symbol < in jedem HTML-Tag encoded werden, nur nicht im `<script>`-Tag, wo es im Kontext von JavaScript als Vergleichsoperator verwendet wird. Um Entwickler in dieser komplexen Problematik zu schulen, bietet OWASP den Developer Guide [vdSO] und Encoder für die verschiedenen Kontexte von HTML [OIML15] an. Allerdings müssen diese Encoder explizit bei der Ausgabe jeder Variablen angewandt werden, wodurch zum einen die Schulung und Sensibilisierung der Entwickler für das Thema Encoding notwendig ist und zum anderen natürlich Fehler durch fehlendes oder falsches Encoding zu Vulnerabilities führen können [SML11]. Oftmals wird diskutiert, ob dieses Encoding mittels Whitelisting oder Blacklisting durch Reguläre Ausdrücke realisiert werden soll. Da jedoch HTML keine reguläre Sprache ist, lässt sich diese mit Regulären Ausdrücken alleine nicht korrekt verarbeiten, sondern Grammatiken müssen verwendet werden [HP05].

Um Injection Vulnerabilities in HTML zu verhindern müssen kontextsensitive Encoder eingesetzt werden, um Eingabedaten in den unterschiedlichen Kontexten der Sprache automatisch korrekt zu encoden [SSS11]. Für die Web-Sprachen HTML und JavaScript lassen sich Injection Vulnerabilities durch die Wahl des richtigen Frameworks zur Sprachverarbeitung verhindern. Allerdings wurde in der Studie von Weinberger et al. [WSA+11] nur in Ctemplate [SG12], welches von Google für die Google Websuche verwendet wird, keine Injection Vulnerability gefunden. Die anderen 13 untersuchten Frameworks enthielten mindestens eine Injection Vulnerability. Für Entwickler entsteht so eine trügerische Sicherheit, wenn sie davon ausgehen, dass das verwendete Framework alle Injection Vulnerabilities verhindert. Diese unterschiedlich guten Encoder sind allerdings sprachspezifisch für HTML und lassen sich nicht für andere Sprachen anwenden, die auch von Injection Vulnerabilities betroffen sind.

Die korrekte Verarbeitung von Eingabedaten weist starke Parallelen zur Verarbeitung von textuellen Modellen zu ausführbarem Code in der modellgetriebenen Softwareentwicklung auf. Modelle entsprechen den Eingabedaten und werden von einem Parser eingelesen, der aus einer Grammatik generiert wird. Bei der Codegenerierung werden Templates eingesetzt [Sch12], da die Erstellung einer Grammatik für die Ausgabesprache zusätzlichen Aufwand bedeutet, sodass auch hier Injection Vulnerabilities existieren. Allerdings wird bei der modellgetriebenen Softwareentwicklung das Modell von einem Entwickler geschrieben, der auch den Codegenerator ausführt, sodass die Ausnutzung dieser Vulnerability dem Entwickler bei einer lokalen Ausführung des Codegenerators keine Vorteile verschafft.

Um XSS in bestehenden Webanwendung mit minimalen Änderungen zu verhindern instru-

mentiert XSS-Guard [BV08] die Anwendung und sendet jede Nutzeranfrage zweimal an die Anwendung: Zuerst das Original und danach die identische Anfrage, bei der allerdings alle Eingabedaten durch sichere Standarddaten ersetzt wurden. Um eine Anfrage zu erkennen, die einen XSS Angriff enthält werden die Antworten der Webanwendung mit einem Browser geparsed und der DOM wird verglichen. Wenn diese DOMs sich unterscheiden, wurde durch die Eingabedaten anscheinend HTML in die Antwort injected. Dieser Ansatz setzt voraus, dass es Standardwerte gibt, auf die die Anwendung genauso reagiert, wie auf echte Eingabewerte. Bei einer Suchanfrage, die unterschiedlich viele Ergebnisse liefert gibt es solche Standardwerte nicht. Und auch XSS, dass in einem mehrschrittigen Dialog erst in der zweiten Antwort auftaucht wird von XSS-Guard nicht erkannt.

Ein eher konstruktiver Ansatz zur Verhinderung von Injection Vulnerabilities besteht darin um eine Sprache, die von einem Programm erzeugt werden soll, wie z.B. SQL eine stark getypte API zu legen, so wie es SQL DOM [MK05] und Java Prepared Statements für SQL machen. Bravenboer et al. haben diesen Ansatz mit StringBorg [BDV07] verallgemeinert, sodass für beliebige Gast-Sprachen wie beispielsweise SQL eine stark getypte API in einer Host-Sprache wie beispielsweise Java generiert werden kann. Außerdem bettet StringBorg die Syntax der Gast-Sprache in die Host-Sprache ein, sodass Entwickler eine fließende Integration der beiden Sprachen nutzen können. Um diese Integration realisieren zu können wird allerdings die Grammatik sowohl der Gast- als auch der Host-Sprache benötigt. Durch die Syntaxeinbettung bestimmt StringBorg den Kontext von Eingabedaten in der Gast-Sprache, allerdings bleibt unklar, wie die Daten encoded werden.

Ein weiterer formaler Ansatz von Samuel et al. [SSS11] definiert und verwendet ein Typsystem für Templatesprachen, die in Webanwendungen verwendet werden. Dieses Typsystem wird verwendet um den Kontext innerhalb von HTML zu bestimmen, an dem Eingabedaten in das Template eingesetzt werden. Entsprechend dieser Kontextinformation wird der korrekte kontextsensitive Encoder für die Eingabedaten aufgerufen. Ein ähnlicher Ansatz aus dem Bereich Unparser von Danielsson [Dan13] beschreibt einen Unparser, der die Korrektheit der Datentypen im ausgegebenen Dokument sicherstellt. Somit kann automatisch das richtige Encoding angewandt werden, dieses muss jedoch separat definiert werden.

Zusammenfassend lässt sich feststellen, dass es Bibliotheken und Frameworks gibt, um Eingabedaten in HTML und SQL korrekt zu verarbeiten. Einige davon müssen manuell im richtigen Kontext der Sprache angewandt werden, andere identifizieren den Kontext und wenden automatisch das korrekte Encoding an. Obwohl das Generieren von Parsern und Unparsern ein lange bekannter Ansatz ist und Typsysteme für Unparser und Templates existieren, so beschreibt doch der in dieser Arbeit entwickelte Ansatz [HKR15] erstmalig wie kontextsensitive Encoder aus kontextfreien Grammatiken abgeleitet werden können.

Dieser Ansatz fügt sich in die grundlegende Auffassung der Language-theoretic Security (LangSec) [SPBS11] ein, dass Shotgun-Parser [BPH13, BDL+14] nicht geeignet sind um Vulnerabilities bei der Verarbeitung von Eingaben zu verhindern. Als Shotgun-Parser werden Programme bezeichnet, die verteilt über den gesamten Programmcode Eingaben parsen und bereits bevor ein Dokument als gültig erkannt wurde anfangen dieses zu verarbeiten. Bei der Verarbeitung wird häufig der Fall missachtet, dass ungültige Dokumente geparsed werden, sodass Angreifer diese nutzen können, um das Programm zu steuern. Die Probleme bei der Verarbeitung von Eingaben

enden jedoch nicht nach dem Parsen, sondern sie stellen auch ein Problem bei der Erstellung von Ausgabedokumenten dar.

Blitzableiter [LR09] von Felix "FX" Lindner verwendet einen Parser um Shockwave Flash (SWF) sicher zu parsen und unparsed anschließend nur solche Dokumente, die auch korrekt erkannt wurden. Schaltet man Blitzableiter vor einen Flash Player, der diverse Vulnerabilities bei der Verarbeitung von Eingaben besitzt, werden maliziöse SWF Inhalte, die diese Vulnerabilities ausnutzen durch das strikte parsen in Blitzableiter verworfen. Mit Blitzableiter teilt der hier beschriebene Ansatz die Idee, dass Eingabedaten nach dem parsen im AST validiert werden, bevor sie ausgegeben werden, wodurch ausschließlich korrekt formatierte Ausgabedokumente erzeugt werden.

8.2. Korrektes Unparsen und Encoden

Um das grundlegende Problem hinter Injection Vulnerabilities wie SQL Injection und XSS zu analysieren, wird nun betrachtet, wie Dokumente einer Sprache zur Kommunikation zwischen Prozessen verwendet werden. Diese Prozesse können dabei verteilt laufen oder auf dem selben Rechner, essentiell ist, dass die sie zur Kommunikation eine Sprache verwenden.

bash

```
1 lars@cyon:~$ ls | wc -l
2 0
3 lars@cyon:~$ touch "a
4 > b"
5 lars@cyon:~$ ls | wc -l
6 2
7 lars@cyon:~$ ls
8 a?b
```

Listing 8.1: Injection in die Ausgabe von `ls`.

Ein Beispiel für eine Injection zeigt Listing 8.1, in dem vier Bash Befehle unter Linux ausgeführt werden. Zuerst werden mit `ls` alle Dateien des aktuellen Verzeichnisses ausgegeben und mittels Pipe an den Befehl `wc -l` weitergegeben. Dieser zählt die Anzahl der Dateien, indem er die Zeilen in der Eingabe zählt und gibt das Ergebnis aus. In diesem Beispiel ist das Verzeichnis zunächst leer. In Zeile 3-4 wird eine Datei angelegt, die einen Zeilenumbruch enthält und in Zeile 5 erneut die Anzahl der Dateien gezählt. Durch den unerwarteten Zeilenumbruch im Dateinamen ist hier das Ergebnis zwei, obwohl nur eine Datei angelegt wurde. Diese Injection in die Ausgabesprache von `ls` ist möglich, da der Zeilenumbruch, welcher als Trennsymbol zwischen Dateien in der Ausgabe von `ls` verwendet wird nicht encoded wird. Das empfangende Programm `wc` kann somit beim Parsen der Dateiliste nicht erkennen, dass es sich nur um eine Datei mit einem speziellen Namen handelt. Interessant ist, dass bei der Ausgabe des Dateinamen an den Benutzer in Zeile 8 der Zeilenumbruch im Gegensatz zur Weiterleitung mittels Pipe als Fragezeichen encoded wird. Dieses Beispiel zeigt, dass jede Ausgabe eines Programms, die von einem anderen Programm verarbeitet wird, als Sprache behandelt werden muss und der

Unparser beim erstellen eines Dokuments dieser Sprache die Steuerzeichen der Sprache in den transportierten Daten encoden muss.

Um diese Sprache korrekt verarbeiten zu können, sodass die Zugehörigkeit von Dokumenten zur Sprache entscheidbar ist, sollte sie durch eine kontextfreie Grammatik definiert werden [SPBS11]. Wenn ein Programm ein Eingabedokument erhält liest es die darin enthaltenen Informationen mittels eines Parsers in einen Parsetree. Dieser Baum wird von irrelevanten syntaktischen Elementen befreit und so in einen AST überführt, welcher von Entwicklern genutzt wird, um die Informationen aus dem Dokument innerhalb der Programmlogik zu verwenden. Die Ausgabe von Informationen verläuft analog, indem der Entwickler Daten, die in der Programmlogik zum Teil aus Eingabedaten berechnet wurden, in den AST einsetzt und der AST durch einen Unparser in ein Dokument ausgegeben wird.

Hier wird der von Danielsson [Dan13] und Arnoldus et al. [AvdBS11] verwendete, Begriff des korrekten (Un)Parse Round-Trip verwendet, der wie folgt definiert ist:

Definition 8.2.1 ((Un)Parse Round-Trip) *Ein (Un)Parsing Prozess für eine Sprache ist ein korrekter Round-Trip, wenn für jeden AST x gilt: $parse(unparse(x)) = x$.*

Im Folgenden wird diese Definition erweitert. Wie im Beispiel von `ls` und `wc` oder HTTP und SQL werden in der Programmlogik Eingabedaten häufig an einen anderen Prozess weitergegeben und dazu in eine Sprache L eingebettet. Ein Angreifer, der dieses Programm mit einem bösartigen Dokument aufruft kann Injection Vulnerabilities in allen Komponenten der Programmlogik ausnutzen, die diese Eingabedaten verarbeiten, sodass die Ausgabe des Programms, welche in einen AST m geschrieben wird, vom Angreifer kontrollierte bösartige Eingabedaten enthält. Die Daten in einem AST m sind bösartig, wenn gilt $\nexists d \in L : parse(d) = m$. Eine Injection Vulnerability in den Kontext von L liegt vor, wenn ein Angreifer ein Dokument in L produzieren kann, welches vom Entwickler nicht vorgesehen ist. Parser und Unparser, die einen korrekten Round-Trip für einen AST m haben, der *bösartige Daten* enthält, verhindern diese Injection Vulnerability durch das korrekte encoden der Daten aus dem AST in ein Dokument und das Dekoden beim Parsen: $parse_{decode}(unparse_{encode}(m)) = m$. Dieser korrekte (Un)Parse Round-Trip nimmt explizit an, dass der AST bösartige Eingabedaten enthält, wohingegen Danielsson und Arnoldus et al. genauso wie die meisten Entwickler davon ausgehen, dass ein AST nur valide Daten enthält, als wäre er durch einen Parser aufgebaut worden.

Abbildung 8.2 verdeutlicht die Notwendigkeit bösartige Eingabedaten im AST beim Unparsen zu behandeln am Beispiel einer Webapplikation. Die Abbildung zeigt, wie ein Template im Webserver zur Initialisierung des ASTs der Ausgabe verwendet wird. Der AST wird durch die Programmlogik noch um zwei weitere Konten erweitert. Der Server parst außerdem eine Eingabe, die von einem Angreifer beeinflusst werden kann und verwendet die erhaltenen Daten, um die Variable `#name#` im AST zu ersetzen. Wird nun ein Unparser verwendet, der einen korrekten Round-Trip für einen AST mit bösartigen Daten unterstützt, ergibt sich das dargestellte HTML Dokument, welches vom Parser des Browsers in den identischen AST übersetzt wird.

Dieser formalisierte Unparsing Prozess findet sich jedoch selten in der Praxis, da er voraussetzt, dass die Ausgabesprache eines Programms zuerst formal definiert werden muss. Template Engines, die einfache String-Ersetzung und -Konkatenation verwenden, benötigen keine Definition der Ausgabesprache, um eine Ausgabe zu erzeugen und werden daher häufig als Vereinfachung

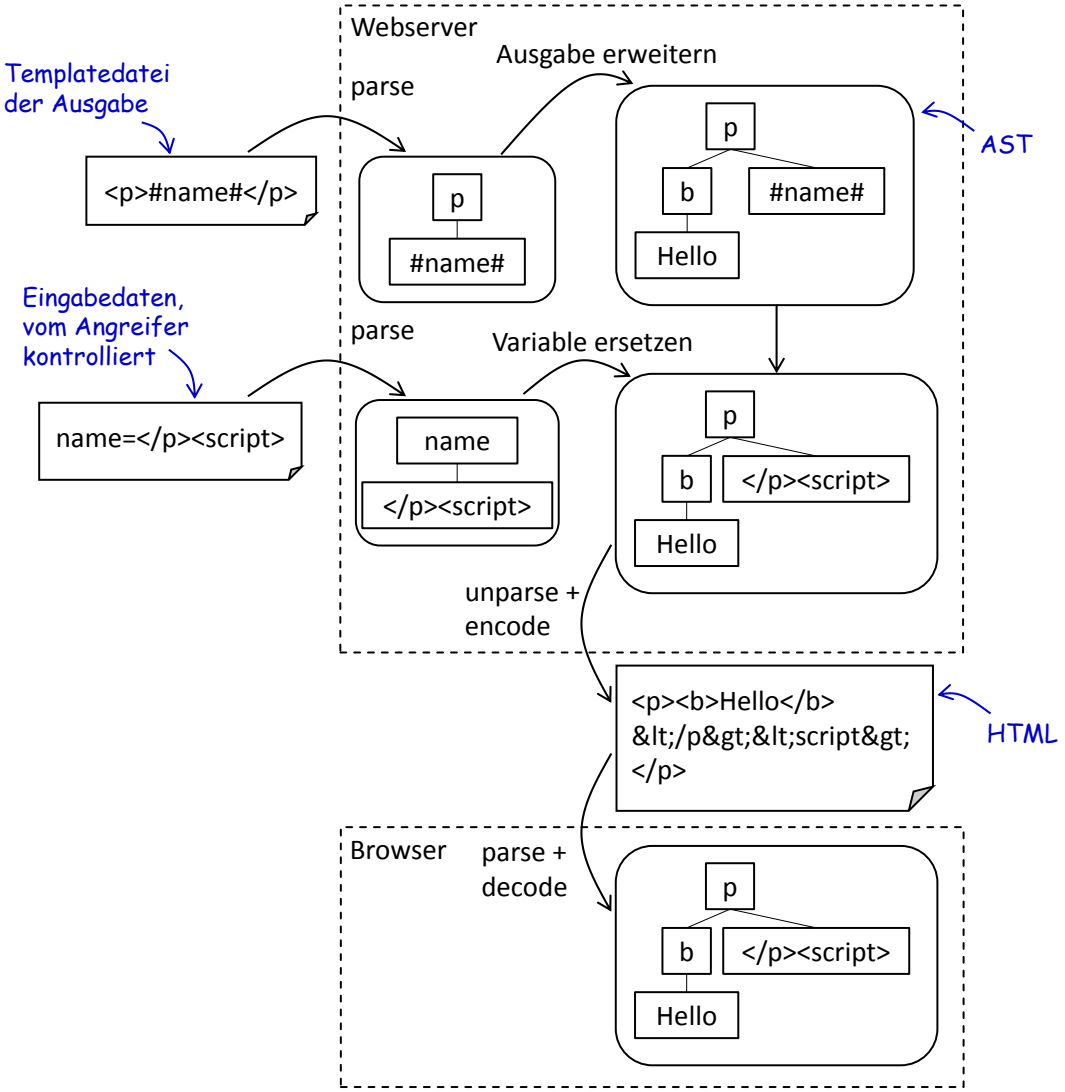


Abbildung 8.2.: Zeigt das Zusammenspiel von encoden und decoden bei der Übertragung eines HTML-Dokuments.

angewandt. Diese einfacheren Ansätze sind jedoch gegen Injection Angriffe anfällig, da Eingabedaten Steuerzeichen der Ausgabesprache enthalten können, die ohne Encodierung in das Ausgabedokument eingesetzt werden. Wird dieses Ausgabedokument geparsed entsteht ein anderer AST, es liegt also eine Injection Vulnerability vor, da der Unparser nicht korrekt encoded hat.

Da unterschiedliche Kontexte einer Sprache verschiedene Steuerzeichen besitzen, benötigt jeder Kontext einen eigenen Encoder. JavaScript muss beispielsweise in einem HTML script-Tag anders encoded werden als in einem onclick-Attribut. Wenn Eingabedaten in einem Template verwendet werden, muss der Entwickler den korrekten Kontext bestimmen, in dem die Daten geschrieben werden und den entsprechenden Encoder verwenden. Wird ein Encoder dabei in einem Kontext verwendet, für den er nicht vorgesehen ist, kann er durch Zufall einige Steuerzeichen encoden. Sobald er jedoch ein Steuerzeichen nicht encoded ist dies eine Injection Vulnerability.

Um Injection Vulnerabilities zu verhindern, müssen Entwickler alle Sprachen definieren, die zur Kommunikation in einem System verwendet werden. Solange das Erstellen einer Grammatik der Ausgabesprachen jedoch wesentlich komplexer ist als Strings zu konkatenieren und in eine Datei auszugeben werden Entwickler weiterhin lieber ohne formale Definition der Ausgabesprache arbeiten. Somit helfen Ansätze, die die Definition von Sprachen und (Un)Parsern während der Entwicklung erleichtern, Injection Vulnerabilities zu verhindern.

8.3. Unparser und Encoder definieren

Die Token einer Grammatik sind entweder Data-Token oder Control-Token [BH13]. Control-Token definieren die Steuerzeichen einer Sprache wie Klammern oder Schlüsselwörter und werden daher auch Keyword genannt. Diese beinhalten keinen vom Nutzer der Sprache frei wählbaren Text. Data-Token beinhalten eben diese frei wählbaren Inhalte wie beispielsweise Variablennamen und -Werte. Dem zuvor beschriebenen (Un)Parsing-Prozess folgend verwenden Entwickler der Programmlogik den AST als Schnittstelle zu einem Dokument einer Sprache und setzen in den Daten-Token des AST die Informationen, die in das Dokument geschrieben werden sollen. Bösartige Eingaben, die darauf abzielen Control-Token in ein Dokument einzufügen, stehen im AST daher in den Data-Token.

Der Reguläre Ausdruck, durch den ein Data-Token in einer Grammatik definiert, wird lässt sich in einem Unparser verwenden, um zu prüfen, ob der Wert des Token im AST die Definition der Grammatik erfüllt. Wird ein Token gefunden, welches Zeichen enthält, die nicht in seiner Definition zugelassen sind, wird das Unparsen abgebrochen, um eine Injection zu verhindern. Dieses Vorgehen verhindert Injection Angriffe, wenn die Regulären Ausdrücke so scharf formuliert sind, dass sie Steuerzeichen explizit ausschließen. Dies ist in der Regel nicht der Fall, da solche strikten Ausdrücke deutlich komplizierter sind und ein Parser für die Sprache auch aus weniger strikten Regulären Ausdrücken generiert werden kann. Außerdem enthalten auch echte Nutzdaten Control-Token, die nicht als solche interpretiert werden sollen. Die Dokumentation von HTML, die selbst in HTML geschrieben ist lässt sich hier als Beispiel nennen.

Um beliebige Nutzdaten in ein Dokument einbetten zu können, werden Escape-Sequenzen verwendet, die Control-Token in einem Data-Token zu encoden. Wie bereits erläutert sind Escape-Sequenzen kontextspezifisch und ein Kontext entspricht einer Token-Definition in der Grammatik.

MCG

```
1 package de.rwth.se.format;
2
3 grammar Container {
4   options {
5     nostring nomlcomments noslcomments
6     noident lexer lookahead = 4
7   }
8
9   Body = LCURLY Element* RCURLY;
10  Element = "tags"
11      LCURLY TagsToken RCURLY;
12  token LCURLY = "{"; token RCURLY = "}";
13
14  encodeTable TagsToken = {
15   "{" -> "&#x0123;", "}" -> "&#x0125;",
16   "&" -> "&#x0038;", " " -> "&#x0020;"
17  };
18
19  subparser token TagsToken =
20    (~('{' | '}' | ' '))+
21  ;
22 }
```

Listing 8.3: Eine MontiCore Grammatik mit integrierter Encoder und Sub-Parser Definition.

Daher erlaubt es der hier vorgestellte Ansatz für jede Token-Definition eine Encoding-Tabelle in der Grammatik zu definieren. Diese Tabelle definiert das kontextspezifische Encoding für diesen Token durch Encoding Regeln, die Control-Token in Escape-Sequenzen übersetzen. Die Beispielgrammatik in Listing 8.3 zeigt, wie die Encoding Table (Zeile 14-17) für das in Zeile 19 definierte Token `TagsToken` direkt in die Grammatik integriert wird.

Die Definition von Encoding Regeln ist ein wichtiger Teil der Sprachdefinition, da Control-Token für die keine Regel existiert oder deren Escape-Sequenz sich zu einem Control-Token kombinieren lässt zu einer Injection Vulnerability in jeder Implementierung führt, die diese Sprachdefinition verwendet. Die Encoding Regeln explizit in einer Encoding Tabelle zu definieren, statt sie in Prosa als Randbemerkung bei der Sprachdokumentation zu erwähnen, hat den Vorteil, dass sich aus diesen Encoding Tabellen ein kontextsensitiver Encoder generieren lässt, der automatisiert vom (Un)Parser aufgerufen werden kann. Dabei encoded der Unparser Data-Token kontextsensitiv und der Parser decoded diese, um einen korrekten Round-Trip auch für einen AST mit bösartigen Daten sicherzustellen.

Um Fehler in der Encoding Table zu erkennen und so Vulnerabilities in den generierten (Un)Parsern zu verhindern, lassen sich die Regulären Ausdrücke heranziehen, mit denen die Token definiert werden. So müssen die Escape-Sequenzen einer Encoding Tabelle durch den Regulären Ausdruck des Token, für den die Encoding Table definiert ist, erkannt werden. Außerdem dürfen die Escape-Sequenzen zusammen mit den durch den Regulären Ausdruck erlaubten Zeichen, die keine Control-Token sind, nicht zu einem Control-Token kombinierbar sein. Dies lässt sich prüfen,

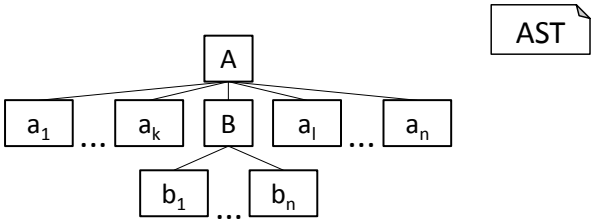


Abbildung 8.4.: Zeigt einen AST der Obersprache A, in dem A mit einer Untersprache B kombiniert wird.

indem alle durch den Regulären Ausdruck erlaubten Wörter in aufsteigender Länge gebildet werden und durch die Encoding Table encoded werden. Ist in diesen encodierten Wörtern ein Control-Token enthalten, wird dieses nicht korrekt encodiert, sodass die Encoding Table eine Injection Vulnerability besitzt. Sind die Escape-Sequenzen länger als die Control-Token, reicht es aus Wörter bis zur Länge des längsten Control-Token zu bilden, um diese Art der Vulnerability zu testen.

Für den Test, ob alle Control-Token in der Encoding Table aufgeführt werden, ist eine Auflistung aller Control-Token der Sprache notwendig. Wenn diese automatisiert aus der Grammatik extrahiert werden können, lässt sich nicht nur die Encoding Table auf Vollständigkeit testen, sondern auch automatisiert generieren, wenn im zu encodierenden Data-Token mindestens zwei Zeichen für die Escape-Sequenz verwendet werden können. Dazu werden die Control-Token eindeutig durchnummeriert und diese Zahl mit den zwei verfügbaren Zeichen binär kodiert. So ergibt sich eine Encoding Table für alle Control-Token. Zusätzlich müssen allerdings alle Pre- und Postfixe von Control-Token encodiert werden. Ein AST, der solche Control-Tokenteile in benachbarten Data-Token enthält würde sonst beim unparsen nicht korrekt encoded und so zu einem inkorrekten (Un)Parse Round-Trip führen. Außerdem wird das erste Symbol der Escape-Sequenz encodiert, damit beim decodieren immer eindeutig entschieden werden kann, ob das Dekodieren notwendig ist.

8.3.1. Sprachkomposition

Bei der Entwicklung von Modellierungssprachen wird Sprachkomposition eingesetzt, um die Effizienz bei der Entwicklung von Sprachen mit Hilfe von Wiederverwendung zu steigern. Wie in Abschnitt 2.3.1 beschrieben wird bietet MontiCore dazu das Konzept der Grammatikerweiterung an. Sprachen, die zur Kommunikation zwischen Prozessen verwendet werden lassen sich ebenfalls mittels Sprachkomposition entwickeln. Allerdings gibt es hierbei die zusätzliche Notwendigkeit Sprachen die unterschiedliche Aspekte beschreiben voneinander zu trennen, um Injections zwischen diesen Aspekten zu verhindern. Beispielsweise beschreibt HTML die Struktur einer Webseite und JavaScript das Verhalten, welches im Browser ausgeführt wird. Allerdings werden beide Sprachen in einem Dokument komponierte verwendet.

Diese Sprachkomposition lässt sich mit Grammatikerweiterung definieren, wobei eine Produktion in der Grammatik der Obersprache A in der Grammatik der Untersprache B erweitert wird und so die unabhängigen Sprachen A und B zu einer Sprache komponiert werden. Ein

beispielhafter AST einer solchen Sprache ist in Abbildung 8.4 dargestellt. Allerdings entspricht nicht jede Grammatikerweiterung einer Trennung in Ober- und Untersprache, da beispielsweise auch wiederverwendbare Sprachteilen durch Grammatikerweiterung erst zu einer sinnvollen Sprache kombiniert werden. Außerdem wird bei der Parsergenerierung aus einer komponierten Sprache in MontiCore ein Parser generiert, der sich aus einem Parser für die Obersprache A und einem für die Untersprache B zusammensetzt. Sobald der Parser für A die Produktion erreicht in die B eingesetzt ist übernimmt der Parser für B die Kontrolle und parsed die Eingabe, bis ein gültiges Dokument von B erkannt wurde. Somit erhält der Parser der Untersprache die Priorität und könnte mehr von der Obersprache parsen, als das Token für das die Untersprache vorgesehen ist.

Um diese Beeinflussungen zwischen den Parsern zu verhindern und einen korrekten (Un)Parse Round-Trip zu erhalten, muss zunächst die Obersprache komplett geparsed werden. Im Anschluss werden Token, die eine Untersprache enthalten, mit dem Parser der Untersprache geparsed. Token, die eine Untersprache enthalten können, werden in der MontiCore Grammatik mit dem Schlüsselwort `subparser` markiert, so wie in Zeile 19 von Listing 8.3 der Token `TagsToken`. In diesem Token der Grammatik `Container` wird mittels Parserkonfiguration die Produktion `Tags` aus Listing 8.5 Zeile 9 eingebettet. In der Grammatik `Container` wird die Encoding Table für den `TagsToken` angegeben (Zeile 14-17), sodass egal welche Sprache in diesen Token eingebettet wird, diese nach der Einbettung nicht die in `Container` definierten Control-Token enthalten.

Für einen korrekten (Un)Parse Round-Trip einer solchen komponierten Sprache ist die Reihenfolge, in der die Unparser der beiden Teilsprachen angewandt werden, entscheidend. Die Anwendung der Encoder beginnt in der untersten (am tiefsten eingebetteten) Untersprache B, die durch ihren Unparser in einen String überführt wird, wobei das in B definierte Encoding angewandt wird. Dieser String wird in den Data-Token des AST der Obersprache A eingefügt, wo B in A eingebettet wird. Daraufhin wird das in A definierte Encoding durch den Unparser auf den AST angewandt, sodass der Data-Token, welcher B enthält erneut encoded wird, wenn er eine Encoding Table in A besitzt. Dieser Encoding Prozess wird für jede Sprache durchgeführt, von der Untersten bis zur Obersten.

Durch diese Reihenfolge des Encodens und Unparsens wird ein Token einer Untersprache so encoded wie durch seine Encoding Table vorgegeben. Im Anschluss wird der gesamte String, der durch das Unparsen der Untersprache entstanden ist, durch die Regeln der Encoding Table der Obersprache encoded, um in den Data-Token der Obersprache zu passen. Somit wird ein Token einer Untersprache während des Unparsens des Gesamtdokuments zuerst vom Encoder der Untersprache und danach von den Encodern der Obersprachen encoded.

Natürlich muss die Reihenfolge, in der die Encoder angewandt werden, mit der Reihenfolge der Decoder übereinstimmen, die während des Parsens des Dokuments angewandt werden. Analog zum Unparsen wird dabei zuerst die Obersprache geparsed und die Token der Obersprache decoded. Token die eine Untersprache enthalten werden nach dem Decoden durch den Parser der Untersprache geparsed. Dieses Decoden übersetzt Escape-Sequenzen wieder zurück in Kontroll-Token der Obersprache, welche wiederum Teil der Untersprache sein können. Somit können Kontroll-Token der Obersprache in ihrer encodeten Form in der Untersprache verwendet werden, ohne den Parser der Obersprache zu beeinflussen. Eine Sprache kann daher beispielsweise in

MCG

```
1  package de.rwth.se.format;
2
3  grammar Tag {
4    options {
5      nostring nomlcomments
6      noslcomments noident
7    }
8
9    Tags = Tag (COMMA Tag)*;
10
11   Tag = LT TEXT GT;
12
13   token LT = "<";
14   token GT = ">";
15   token COMMA = ",";
16   token TEXT =
17     (
18       ~('<' | '>' | '\\' | ',')
19     |
20       (
21         '\\'
22         ('<' | '>' | '\\' | ',')
23       )
24     )+
25   ;
26
27   encodeTable TEXT = {
28     "\\" -> "\\\\",
29     ","  -> "\\,",
30     ">"  -> "\\>",
31     "<"  -> "\\<"
32   };
33 }
```

Listing 8.5: Eine MontiCore Grammatik mit integrierter Encoder Definition.

einen ihrer eigenen Data-Token eingebettet werden, allein durch die korrekte Definition der Encoding Table dieses Token.

8.4. Anwendung von Unparsern und Encodern

Im Folgenden wird untersucht, wie die zuvor beschriebene Definition von Sprachen zusammen mit ihrem kontextspezifischen Encoding sich in Anwendungen einsetzen lässt, die komplexe Eingaben zulassen oder touringvollständige Sprachen ausgeben.

8.4.1. Sprach Features reduzieren

In den vorhergehenden Abschnitten werden Injections betrachtet, die die Struktur des Gesamtdokuments verändern, welches auch als 'injecting up' [Ope15a] bekannt ist. Nun wird zusätzlich das 'injecting down' näher untersucht, bei dem Control-Token in einen Ausdruck injectet werden, mit der Absicht die Semantik des Ausdrucks zu verändern, ohne aus dem Token auszubrechen.

Eine naive Implementierung einer Webapplikation, die es Nutzern erlaubt Teile der Ausgabe selbst zu definieren könnte beispielsweise HTML in einem Forum oder Wiki Eintrag erlauben und diese Nutzereingaben direkt in die Ausgabe übernehmen. Dies entspricht einer 'injecting down' XSS Vulnerability, da ein Angreifer auch JavaScript einfügen kann, welches ungeprüft in die Ausgabe übernommen wird. Um dies zu verhindern, muss die Nutzereingabe geparsed und überprüft werden, um Token der Sprache zu eliminieren, die nicht in die Ausgabe gelangen sollen. Es wird also ein Filter benötigt, der unerwünschte Token aus der Eingabe entfernt, so wie eine Firewall unerwünschten Netzwerkverkehr filtert.

Für dieses Problem gibt es mindestens zwei Lösungen: Zum einen kann die Grammatik der Ausgabesprache so reduziert werden, dass sie nur eine Teilmenge der Original Ausgabesprache erlaubt, die keine unerwünschten Token zulässt. Somit ist es in der Anwendung nicht möglich ein Ausgabedokument zu erstellen, welches das unerwünschte Token enthält. Dieser Ansatz eignet sich, wenn ein Sprachelement komplett verboten werden soll, sodass weder Entwickler noch Nutzer dieses verwenden können. Eine Webanwendung, in der das `script`-Tag in keiner Webseite verwendet werden darf ist ein Beispiel, wo diese Methode zielführend ist. Allerdings müssen Entwickler typischerweise alle Sprachelemente der Ausgabesprache verwenden und nur in bestimmten Bereichen, wie z.B. einem Kommentar eines Besuchers sind nicht alle Sprachelemente erlaubt. Daher muss die Grammatik so umgeschrieben werden, dass die reduzierte Sprache nur in bestimmten Token gilt, wie beispielsweise in HTML-Tags mit einem speziellen Namen. Die eigentliche Reduzierung der Sprache kann durch eine Anpassung der Produktionen der Grammatik oder eine Encoding Table für einen spezifischen Token realisiert werden. Die Grammatik der Ausgabesprache basierend auf Namen oder Eigenschaften, die in einem Ausgabedokument verwendet werden, anzupassen ist allerdings weder elegant noch einfach.

Zum anderen kann eine Grammatik zum Parsen der reduzierten Sprache definiert werden, die bei der Eingabeverarbeitung und damit vor dem Unparsen verwendet wird, um Eingabedaten zu erkennen, die zur reduzierten Sprache gehören. Diese überprüften Eingabedaten werden aus dem AST des Eingabedokuments direkt in den AST des Ausgabedokuments kopiert. Dabei verhindert die Eingabesprache, welche eine reduzierte Version der Ausgabesprache ist, ein 'injecting down' in die Ausgabesprache und die Unparser und Encoder der Ausgabesprache verhindern ein 'injecting up'.

Beide Ansätze verhindern Injection von unerwünschten Sprachelementen in die Ausgabe, wenn sie korrekt implementiert sind. Die Grammatik der Ausgabesprache zu reduzieren ist die zuverlässigere Methode, da alle Daten die ausgegeben werden, egal aus welchen Eingabedaten sie stammen, korrekt durch den Unparser in das Dokument ausgegeben werden. Allerdings ist es einfacher eine reduzierte Sprache zu spezifizieren, die zur Erkennung der Eingabe verwendet wird. Daher gilt es abzuwägen, welche Variante für ein System geeignet ist: Für Systeme, die die Eingabe direkt in der Ausgabe verwenden ohne diese zu verändern, reicht es aus die Eingabe zu erkennen. Im Gegensatz dazu ist bei Systemen, die Eingaben aus verschiedenen Quellen

in einer komplexen Verarbeitungslogik verändern oder anreichern, eine reduzierte Ausgabesprache unumgänglich, da allein durch die Validierung der Eingabedaten nicht sichergestellt werden kann, dass durch die Verarbeitung der Eingaben keine unerwünschten Teile der Ausgabesprache erzeugt werden. Grund hierfür ist, dass bei der Überprüfung der Eingabe unklar ist, in welcher Ausgabesprache und in welchem Kontext dieser Sprache die Eingabe verwendet wird. Wird die Eingabe in verschiedenen Sprachen oder Kontexten verwendet, ist eine korrekte kontextspezifische Encodierung vor dem Unparsen sogar unmöglich.

8.4.2. Encoding für ausführbare Sprachen

Sprachen, die es erlauben touringvollständige Berechnungsvorschriften zu spezifizieren, zur Kommunikation zu verwenden ist problematisch, da nicht-triviale Eigenschaften wie beispielsweise die Terminierung des spezifizierten Verhaltens nicht bei der Eingabevalidierung entschieden werden können [BDL+14]. Auch bei der Erzeugung von Dokumenten einer solchen Sprache L_1 durch einen Unparser besteht dieses Problem. Im Folgenden sei L_0 die Eingabesprache eines Prozesses, der Dokumente in einer touringvollständigen Sprache L_1 ausgibt. Bei der Ausführung von L_1 erzeugt dieses Programm Dokumente in der Sprache L_2. Der Unparser kann zwar den Kontext innerhalb der Sprache L_1 erkennen und Eingaben aus der Sprache L_0 dementsprechend so encoden, dass die Semantik des in L_1 spezifizierten Programms nicht verändert wird, jedoch ist es nicht möglich vorauszuberechnen in welchen Kontexten der Ausgabesprache L_2 des Programms eine Eingabe aus L_0 verwendet wird.

Im Falle einer Webapplikation sind die Parameter, die zur Erzeugung einer Webseite verwendet werden, aus der Sprache L_0. Das vom Unparser auf dem Webservers generierte Dokument, welches HTML und JavaScript enthält, gehört zur Sprache L_1. Und L_2 ist die Ausgabe, also der DOM, der bei der Ausführung des JavaScript im Browser entsteht.

Wird eine touringvollständige Sprache (wie JavaScript zusammen mit HTML) zur Kommunikation verwendet, ist die Verhinderung von Injection Vulnerabilities in die Ausgabe, die bei der Ausführung der Sprache entsteht (der DOM), allein durch den Sender der Nachricht (den Webserver) nicht möglich. Dies liegt daran, dass die Ausführungs-Semantik des vom Sender generierten Dokuments nicht bereits während der Generierung mit betrachtet werden kann.

Ein korrekter (Un)Parse Round-Trip stellt sicher, dass Informationen bei einem Empfänger so ankommen, wie sie der Sender abgeschickt hat und keine Eingaben aus L_0 den L_1-Parser des Empfängers verwirren können. Wenn allerdings nach dem Parsen dieses L_1-Dokuments dieses als Programm ausgeführt oder interpretiert wird, wie im Falle von JavaScript, muss durch den Interpreter und das Programm selbst sichergestellt werden, dass Eingaben aus L_0, die zur Generierung des Programms verwendet werden, der Programmausgabesprache L_2 entsprechend korrekt encoded werden.

Um dieses Encoding nach L_2 automatisiert korrekt durchzuführen, kann ebenfalls der zuvor vorgestellte Ansatz verwendet werden, um kontextsensitive Encoder für die Programmausgabesprache L_2 (im Beispiel der DOM) zu generieren. Da der Encoder in der Sprache L_1 (also JavaScript im Beispiel) ausgeführt wird, muss seine Generierung allerdings auf L_1 angepasst werden und die Möglichkeiten zur Ausgabe von Dokumenten, die in L_1 zur Verfügung stehen, genutzt werden.

Das korrekte Encoding und Unparsen könnte auch komplett durch die Ausführungsumgebung von L_1 durchgeführt werden. Diese müsste als einzige Ausgabemöglichkeit einen AST des Ausgabedokuments zur Verfügung stellen und die Ausgabesprache fest vorgegeben bzw. erzwingen, dass diese definiert wird, bevor eine Ausgabe in dieser Sprache möglich wird. Im zuvor genutzten Beispiel bedeutet dies für Browser, dass sie bei Schreibzugriffen aus JavaScript auf bestehende DOM-Elemente das entsprechende Encoding für diese Elemente anwenden müssen.

HTML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head> <title>Example Page</title> </head>
  <body>
    <form method="GET" action="#actionURL#">
      <label for="input"> Name: </label>
      <input type="text" id="input" name="name" value="#name#" />
      <input type="submit" value="Register"/>
    </form>
    <div>
      <p>#name#</p>
      <button onclick="alert(&quot;#name#&quot;);">Test1</button>
      <button onclick="alert('#name#');">Test2</button>
      <input type="text" name="input" value="#name#" />
      <script>
        var name ="#name#" + '#name#';
      </script>
    </div>
  </body>
</html>
```

Listing 8.6: Ein HTML Template mit den Variablen `name` und `actionURL`.

8.5. MontiCoder

Die beschriebene Generierung von kontextsensitiven Encodern und Decodern aus Encoding Tables wurden im Rahmen der Arbeit von Stephan Kugelmann [Kug15] in MontiCoder, einer Erweiterung von MontiCore, umgesetzt. Die Encoder wurden in die von MontiCore generierten Pretty-Printer integriert und die Decoder in die generierten Parser.

Dabei hat sich gezeigt, dass die kompakte Schreibweise des Encodings in der MontiCore Grammatik für die Definition neuer Sprachen geeignet ist, jedoch unnötig umständlich wird, wenn Encoding Funktionen bereits existieren oder das Encoding sich einfacher durch Berechnungen in Programmcode definieren lässt. Für diese Fälle ermöglicht es MontiCoder die generierten En- und Decoder für spezifische Token durch Java Methoden zu ersetzen.

Die korrekte Definition des Encoding ist essentiell, um Injection Vulnerabilities zu verhindern. Daher sind Qualitätssicherungsmaßnahmen für diese Definition besonders sinnvoll. Aber auch

ohne zusätzliche Testfälle lässt sich allein durch die Sprachdefinition während des unparsens prüfen, ob das Encoding der Grammatik entspricht. Dazu wird der encodierte Wert gegen den Regulären Ausdruck geprüft, welcher in der Grammatik das encodierte Token definiert. Wenn der Reguläre Ausdruck den Wert nicht akzeptiert wird die weitere Verarbeitung abgebrochen, um eine Injection zu verhindern. Diese Konsistenzprüfung ist besonders wirkungsvoll, wenn der Reguläre Ausdruck strikt definiert ist und somit keine Kontroll-Token erlaubt.

Um mit MontiCoder eine Ausgabe zu erstellen, muss zunächst der AST des Ausgabedokuments aufgebaut werden. Soll ein großes Dokument mit wenigen Variablen ausgegeben werden, ist viel zusätzlicher Programmcode notwendig, um dieses Basisdokument als AST zu instantiieren. Um diese Instanziierung zu vereinfachen wurde ein Template Mechanismus implementiert, mit dem ein Basisdokument aus einer Datei geparsed werden kann, sodass der AST mit der Struktur des Basisdokuments befüllt wird. Um einen validen AST zu erhalten, muss das Basisdokument parsebar sein, sodass ein AST entsteht, der zur Bestimmung der Kontexte der Encoder herangezogen werden kann.

Das Basisdokument kann Variablen enthalten, die wie in Zeile 6 und 8 in Listing 8.6 durch # markiert werden. Diese Variablen werden im AST durch vom Entwickler spezifizierte (Eingabe-) Daten ersetzt. Bevor der AST durch den Unparser ausgegeben und encoded wird, kann der Entwickler noch beliebige Kontroll- und Data-Token hinzufügen und diese mit Daten füllen. Kontroll-Token müssen dem AST als neue Knoten hinzugefügt werden, da alle Kontroll-Token, die in einem Data-Token (durch Eingabedaten) eingefügt werden, kontextspezifisch encoded werden.

Kapitel 9.

Modellgetriebene Entwicklung von Privacy

Durch die Einbettung von Cyber Physical Systems (CPS) in das Umfeld seiner Nutzer entstehen Daten, die tiefe Einblicke in das Privatleben der Nutzer erlauben. Diese wecken Begehrlichkeiten bei Serviceanbietern, die Nutzerdaten abseits der Serviceerbringung zu nutzen beispielsweise um den Nutzern zielgruppenspezifische Werbung für Produkte und Dienstleistungen Dritter anzuzeigen. Aber insbesondere bei diesen Dritten, die am eigentlichen Service nicht beteiligt sind, besteht Interesse Nutzerdaten zu verwenden beispielsweise um lohnenswerte von nicht lohnenswerten Kunden zu unterscheiden [ZGMW14, ISKC11, TJA10, HHW13]. So könnte beispielsweise eine Versicherung vor Vertragsabschluss das Verhalten ihrer neuen Kunden, welches durch ein CPS in großen Teilen nachvollziehbar ist, analysieren und nur denen einen Vertrag anbieten, die aufgrund ihres Verhaltens die Versicherung nicht in Anspruch nehmen und so ihren Gewinn maximieren [Cou13]. Denkbar sind Szenarien, bei denen anhand von Sensoren im Auto das Fahrverhalten beobachtet wird und dementsprechend Rabatte bei der KFZ-Versicherung gewährt werden. Um attraktive Versicherungskonditionen zu erhalten müsste der Kunde sein Verhalten ändern, was wiederum durch das CPS für die Versicherung beobachtbar wird.

Damit solche Szenarien, in denen durch technische Systeme Kontrolle über das Verhalten von Menschen ausgeübt wird nicht eintreten, müssen verschiedene Perspektiven auf das Problem Privacy betrachtet werden. Im Folgenden werden dazu zunächst die Nutzerbedenken formuliert und im Anschluss die Schwierigkeiten von Serviceanbietern bei der Entwicklung von Systemen, die diese Bedenken berücksichtigen.

9.1. Privacy-Bedenken von Nutzern

Wenn Daten bei der Verarbeitung in einem CPS außerhalb der Einflusssphäre des Nutzers verarbeitet werden, beispielsweise in der Cloud [HHCW12, HHM+13, EHH+14, HHMW14, HNPR14, BdDPP14], entsteht beim Nutzer der Eindruck die Kontrolle über seine Daten zu verlieren [TJA10, HHW13]. Darum müssen Serviceanbieter von cloud-basierten Services die Geheimhaltung und den Schutz der Nuterdaten bei Speicherung und Verarbeitung zusichern und einhalten. Anderenfalls könnte aufgrund von Privacy-Bedenken die Benutzerakzeptanz des Services schwinden. Diese Privacy-Bedenken entstehen, da der Nutzer faktisch keine Kontrolle über die Zugriffe auf seine Daten *hat*, wenn diese in der Cloud gespeichert sind, sodass Daten an Dritte weitergegeben werden können oder eine Nutzung zu anderen als den ursprünglich vorgesehenen Zwecken möglich ist [HHW13]. Aufgrund dieser Bedenken tendieren Nutzer dazu cloud-basierte Services und CPS für (sehr) sensible Daten, wie beispielsweise Gesundheitsdaten

abzulehnen [LHL15].

Aus einer Nutzerbefragung [Smi12] geht hervor, dass Nutzer einen angemessenen Schutz ihrer Daten erwarten, die Einhaltung von Gesetzten durch den Serviceanbieter fordern und über die Nutzung ihrer Daten informiert werden wollen. Dies unterstreicht, wie wichtig es ist den Nutzer mit in die Lösung des Privacy-Problems einzubeziehen, um seine individuellen Privacy-Anforderungen zu erfüllen. Insbesondere muss Anforderung A6 im Kontext Privacy erfüllt werden, indem dem Nutzer transparent dargelegt wird, wie und zu welchem Zweck seine Daten verarbeitet und gespeichert werden. Nutzer geben ebenfalls an, dass sie persönliche Daten wie Konto und Kreditkartennummern im Internet so sparsam wie möglich preisgeben [EHKR14]. Hierbei scheint jedoch das in einer Befragung angegebene und das tatsächliche Verhalten, ähnlich wie in Belangen des Umweltschutzes, voneinander abzuweichen. Den Befragten ist anscheinend klar, dass die Weitergabe von personenbezogenen Daten immer ein gewisses Risiko darstellt, daher geben sie in einer Befragung an keine Daten weiterzugeben [EHKR14]. Allerdings geben Nutzer ihre Daten weiter, sobald ihnen ein gewisser Anreize wie beispielsweise ein Rabatt in einem Onlineshop oder die kostenlose Nutzung eines Dienstes geboten wird [Les12]. Diese Anreize bewertet jeder Nutzer individuell anders, sodass Privacy-Anforderungen an ein System immer nutzerspezifisch sind und es somit schwierig ist *die individuellen* Privacy-Anforderungen aller Nutzer beim Entwurf eines Systems herauszufinden und zu berücksichtigen.

Die Anreize seine Daten einem System zur Verfügung zu stellen können sich zusätzlich über die Zeit verändern. Gerät beispielsweise ein Nutzer eines Systems, welches Vitalzeichen mit Sensoren überwacht in eine medizinische Notsituation, die er ohne fremde Hilfe wohl nicht überleben wird, ist der Nutzer sicherlich bereit seine Privacy komplett aufzugeben und jedem Zugriff auf seine Gesundheitsdaten zu geben, wenn er dadurch am Leben bleibt [Bec03]. Dies zeigt, dass Privacy keine feste Anforderung an ein System ist, die sich für alle Nutzer und all ihre Lebenslagen fix implementieren lässt, sondern vielmehr Nutzer die Möglichkeit haben müssen individuelle Privacy-Entscheidungen während der Laufzeit des Systems zu treffen.

9.2. Herausforderungen bei der Realisierung von Privacy

Die Privacy von Nutzern stellt auch Serviceanbieter vor Herausforderungen bei der Implementierung und dem Betrieb von cloud-basierten Diensten. Denn wenn Serviceanbieter die zuvor beschriebenen Bedenken ihrer Nutzer nicht adressieren, sondern diese ignorieren müssen sie damit rechnen dass die Nutzer den Dienst nicht akzeptieren und daher nicht verwenden oder kostenintensive Gerichtsverfahren gegen den Serviceanbieter angestrengt werden, wenn er geltenden Gesetze nicht einhält [Pea09, ZGMW14].

Diese gesetzlichen Vorgaben betreffen beispielsweise den Speicherort von Daten, sodass nach EU Recht Steuerdaten und Bilanzen eines Unternehmens innerhalb der EU gespeichert werden müssen [De 13]. Bei der Speicherung dieser Daten in der Cloud gibt es jedoch nicht bei jedem Cloudservice eine Garantie, dass diese Bedingung eingehalten wird, sodass Serviceanbieter sich strafbar machen könnte, wenn sie den Dienst nutzen.

Eine weitere gesetzliche Anforderung, das sogenannte “right to be forgotten” [Ros12] verlangt vom Serviceanbieter, dass alle Nutzerdaten nach einer bestimmten Zeit automatisiert gelöscht werden können. Diese Gesetze stellt Serviceanbieter vor die Aufgabe eine gesetzeskonforme Um-

setzung zu realisieren, was besonders bei Cloud Technologien eine Herausforderung darstellt, bei denen zur Kosteneinsparung Dienste Weltweit verteilt erbracht werden [Ros12, HHW13]. Da der Serviceanbieter seinen Service üblicherweise auf einer von einem Cloud-Provider bereitgestellten Plattform realisiert, ist der Serviceanbieter nicht nur daran interessiert die Privacy-Bedürfnisse seiner Nutzer zu erfüllen, sondern ist selber wieder Nutzer eines Dienstes, bei dem er je nach der Art der verarbeiteten Daten besondere Anforderungen an die bereitgestellte Plattform hat.

Bei der Serviceentwicklung entscheidet der Servicebetreiber über die Funktionen, die den Nutzern des Services zur Verfügung stehen sollen und der Serviceentwickler definiert daraufhin, wie die Daten verarbeitet werden, damit die Funktionalität erbracht wird. Dabei lassen sich die individuellen Privacy-Bedürfnisse einzelner Nutzer zwar erheben, da diese jedoch in sich widersprüchlich sind da Nutzer-Privacy unterschiedlich bewerten, ist es schwer einen Service anzubieten, der die Privacy-Bedenken aller Nutzer befriedigt. Daher wird ein Service mit fixer Funktionalität entwickelt und die Verarbeitung von Nutzerdaten in einer Privacy Policy beschrieben. Diese Privacy Policy wird von Juristen formuliert, die sich von den Serviceentwicklern beschreiben lassen wie die Daten verarbeitet werden. Eine einmal von der Rechtsabteilung eines Unternehmens abgesegnete Privacy Policy wird selten geändert, da der Veränderungsprozess der Privacy Policy, der von den Entwicklern über die Rechtsabteilung bis zu den Nutzern abgestimmt werden muss sehr aufwändig ist. Ein solches Vorgehen lässt sich nicht direkt in einen agilen Entwicklungsprozess integrieren, so wie Anforderung A1 dies fordert. Da die menschliche Kommunikation fehleranfällig ist [GF14], entstehen Missverständnisse zwischen Entwicklern und der Rechtsabteilung, was bei der Erstellung der Privacy Policy berücksichtigt wird und dazu führt, dass eine sehr defensive Privacy Policy entsteht, die Nutzerklagen auch dann standhält, wenn die tatsächliche Nutzerdatenverarbeitung von der abweicht, die die Entwickler formulieren wollten und die Rechtsabteilung verstanden hat [PAS02]. Eine solche ungenaue, nicht aktuelle Beschreibung der Datenverarbeitung führt zu Unsicherheit und Intransparenz für die Nutzer.

Zusammenfassend lässt sich feststellen, dass durch die individuellen Privacy-Anforderungen unterschiedlicher Nutzer die Datenverarbeitung eines Services zur Laufzeit durch den Nutzer anpassbar sein muss und dies einen Komplexitätszuwachs in der Entwicklung von Cloud-basierten Services bedeutet. Der in dieser Arbeit vorgestellte modellgetriebene Privacy Ansatz unterstützt Entwickler dabei Services zu entwickeln, die die individuellen Privacy-Anforderungen der Nutzer respektieren und hilft dabei die Vorteile von Privacy-by-Design in den agilen Entwicklungsprozess zu bringen.

9.3. Anwendungsszenario und existierende Lösungen

Um die Bedenken der Nutzer und die Herausforderungen der Serviceanbieter und -Entwickler zu konkretisieren, wird im Folgenden zunächst ein Beispielszenario aus dem Projekt Intelligent Privacy-aware Cloud-based Services (IPACS) vorgestellt und im Anschluss mögliche Lösungen mit bestehenden Ansätzen diskutiert. In dem Beispielszenario *Public Mobility Assistance* wird ein Dienst für eine durch den Demografischen Wandel wachsende Nutzergruppe, die alleinstehenden Rentnerinnen, entwickelt. Der Service soll eine Rentnerin, die in ihrem eigenen Haus wohnt, dabei unterstützen trotz ihrer altersbedingten Beschwerden ihren bisherigen Lebenswandel und ihre Autonomie zu behalten, indem er sie gezielt in ihrer Mobilität unterstützt.

Über das Nutzerinterface, welches die Nutzerin mit sich führt werden verschiedene Informationsquellen integriert, um sie beispielsweise bei ihrem Weg zum Supermarkt mit öffentlichen Verkehrsmitteln zu unterstützen und ihr die Sicherheit zu geben entsprechend ihrer tagesaktuellen Verfassung ihr Ziel zu erreichen, jederzeit eine Pause machen zu können, hilfsbereite Mitmenschen zu finden oder sich im Notfall von ihrer Familie abholen zu lassen.

Um einen solchen Service zu realisieren, müssen teils sensible Informationen wie die Gesundheitsdaten der Nutzerin, ihre Gewohnheiten und ihr Aufenthaltsort durch den Service verarbeitet werden. Damit die Nutzerin die Kontrolle über ihre Daten behält, werden alle Daten, die den von ihr kontrollierten Bereich, also beispielsweise ihre Wohnung verlassen vor der Übertragung in die Cloud mit dem im Projekt IPACS entwickelten Ansatz verschlüsselt [HHK+16, HHK+14]. Ein Service hat somit nicht automatisch Zugriff auf alle Daten, sondern die Nutzerin muss diesen explizit erlauben. Bei Sensorwerten wie beispielsweise Gesundheitsdaten ist es dabei möglich den Zugriff auf ein bestimmtes Zeitintervall oder für die aktuell gemessenen Werte freizugeben. Die Nutzerin ist also grundsätzlich in der Lage Zugriffe auf ihre Daten, die in der Cloud gespeichert sind zu kontrollieren.

Um Berechnungen auf diesen Daten durchzuführen, müssen sie durch den Service entschlüsselt und verarbeitet werden. Damit der Nutzerin transparent nachgewiesen werden kann (Anforderung A6), dass der Service ihre Privacy respektiert ist es notwendig bereits während der Entwicklung die Prinzipien Privacy-by-Design [Cav16] und Privacy-by-Policy [SC09] zu befolgen, sowie Spezielle Hinweise für die Entwicklung von Cloud Services umzusetzen [Pea09]. Allerdings sind diese Prinzipien vage, sodass es schwierig ist direkt eine Implementierung zu erstellen, die diese umsetzt [GTD11, MdAY14]. Zur Umsetzung von Anforderung A5 eignen sich diese Prinzipien daher nicht, sondern den Serviceentwicklern müssen konkrete Werkzeuge und Methoden bereitgestellt werden.

Um die individuellen Nutzeranforderungen an die Datenverarbeitung einem Service vor der Nutzung mitzuteilen gibt es verschiedene Ansätze aus dem Bereich der Zugriffskontrolle. Karjoth et al. [KS02] entwickelten ein Privacy Policy Model, welches in einem Unternehmen dazu genutzt werden kann die Zugriffskontrolle so umzusetzen, dass die Nutzerzustimmung bei einem Zugriff auf Nutzerdaten sichergestellt wird und das Unternehmen alle als notwendig definierten Schritte durchführt, wenn ein Zugriff geschieht. Die Erweiterung P-RWBAC des Role Based Access Control Schemas [He03, NBL+10] erlaubt es Zugriffsrechte nicht nur auf Basis von Rollen zu definieren, sonder auch den Zweck und die notwendigen Schritte welche bei einem Zugriff durchgeführt werden müssen in die Entscheidung über einen Zugriff miteinzubeziehen.

Der Zweck einer Datenverarbeitung wird Nutzern in einer Privacy Policy dargestellt. Zur Erstellung einer Privacy Policy lässt sich Server Privacy ARchitecture and CapabiLity Enablement (SPARCLE) [BKKF05] heranziehen, welches Werkzeuge bereitstellt, um eine Privacy Policy, die in natürlicher Sprache nach einem bestimmten Schema geschrieben ist in XACML [OAS13], einem Standard zur Definition von Zugriffsrechten, oder P-RBAC zu übersetzen. Dabei ist eine Übersetzung in beide Richtungen möglich, sodass formalisierte Zugriffsrechte auch wieder in natürlicher Sprache übersetzt werden können. Im Rahmen einer Studie mit Nutzern, die eine Privacy Policy erstellen, Auditoren und Managern wurde gezeigt, dass SPARCLE für diese Nutzergruppen eine starke Erleichterung ihrer Arbeit darstellt [BKK06]. Entwickler, die statt natürlicher Sprache oder XML eine eher kompakte Sprache zur Beschreibung bevorzugen

[KKP+09], wurden in der Studie jedoch nicht betrachtet. Die PrimeLife Privacy Policy Language (PPL) [TSR11], eine Erweiterung von XACML ermöglicht es Nutzern und Service Providern eine Privacy Policy zu definieren. Möchte ein Nutzer einen Service verwenden überprüft ein spezieller Client, ob die Privacy Policy des Services mit der des Nutzers kompatibel ist und fragt den Nutzer, ob er seine Privacy Policy anpassen möchte, wenn dies nicht der Fall ist.

All diese Ansätze lassen sich verwenden um eine Privacy Policy zu definieren, allerdings eignen sie sich nicht, um sie in einen agilen Entwicklungsprozess zu integrieren, wie dies von Anforderung A1 gefordert wird. Zur Spezifikation von Privacy-Eigenschaften während der Entwicklung wurde die Privacy-aware Modeling Language (PaML) [CF12] konzipiert, die UML mittels Stereotypen erweitert, um diese Eigenschaften zu integrieren. Mittels Atlas Transformation Language (ATL), welche in OCL eingebettet ist, werden Regeln definiert nach denen Daten verarbeitet werden dürfen. Diese Regeln lassen sich durch Werkzeuge gegen die PaML Modelle prüfen und stellen so sicher, dass einmal definierte Regeln auch während der Weiterentwicklung des Systems eingehalten werden. Um diese Anforderungen an Privacy identifizieren und festhalten zu können erweiterten Jutla et al. [JBA13] UML Use Case Diagramme mit Techniken, die die Privacy fördern, wie `Notice and Agreement`, `Pseudunymization` und `Anonymization`. Diese Ansätze nehmen an, dass Nutzer so in den Entwicklungsprozess eingebunden werden können, dass sie ihre Privacy-Anforderungen in die Entwicklung einbringen können.

Auch wenn das zuvor beschriebene Szenario nur eine Nutzerin erwähnt, lohnt sich der Betrieb eines Cloud Services nur, wenn er von vielen Nutzern genutzt wird, die nicht an seiner Entwicklung beteiligt sind [AFG+10]. Somit können in diesen Ansätzen nicht die individuellen Privacy-Anforderungen aller Nutzer bereits während der Entwicklung erfasst und im System entsprechend umgesetzt werden. Um die Akzeptanz aller Nutzer zu erreichen muss stattdessen die Entscheidung über Privacy dem Nutzer zur Laufzeit des Systems überlassen werden. Diese hohe Flexibilität bezüglich der möglichen Eingaben erhöht unweigerlich die Komplexität des Entwurfs und der Realisierung eines solchen Services, welche die Entwickler beherrschen müssen.

9.4. Privacy Development Language

Um Serviceentwickler dabei zu unterstützen das abstrakte Prinzip Privacy-by-Design umzusetzen und ihnen zu ermöglichen ihre Privacy-Überlegungen direkt während der Erstellung des Datenmodells eines Services zu beschreiben wird die Privacy Development Language (PDL) eingeführt. Diese verbindet bereits in der Entwicklung das objektorientierte Datenmodell mit der Beschreibung der Privacy-Eigenschaften des Services. Ein PDL-Modell representiert die Datenstruktur der Nutzerdaten, die durch einen Service verarbeitet werden zusammen mit Informationen über deren Nutzung innerhalb des Services. Diese Informationen werden verwendet, um Endnutzern möglichst viele Details über die Nutzung ihrer Daten durch den Service zur Verfügung zu stellen und ihnen somit eine fundierte, informierte Entscheidung über ihre Privacy Configuration zu ermöglichen.

Im Folgenden Abschnitt wird zunächst die PDL und ihr Einsatz im modellgetribenen Entwicklungsprozess aus der Entwicklerperspektive beschrieben, bevor weiter auf den Benutzer und die ihm durch den modellgetriebenen Privacy Ansatz zur Verfügung gestellten Informationen eingegangen wird.

In Pearsons “Top Sechs für Software Entwickler” [Pea09] werden die wichtigsten Maßnahmen genannt, die ein System umsetzen bzw. bereitstellen soll, um den Schutz der Privatsphäre seiner Nutzer zu gewährleisten. Durch den modellgetriebenen Privacy Ansatz in Verbindung mit der in UPECSI [HHK+14] eingesetzten kryptographischen Zugriffskontrolle, die Zugriff auf Nutzerdaten unabhängig von deren Speicherort nur ermöglicht wenn der Nutzer dies zuvor autorisiert hat werden Pearsons “Top Sechs” umgesetzt. Um dies zu erreichen muss ein Service dem Nutzer eine Privacy Policy sowie Data Usage Monitoring bereitstellen. Außerdem soll eine Auditierung der Serviceimplementierung zeigen können, dass die in der Privacy Policy versprochenen Eigenschaften eingehalten werden. Diese Maßnahmen sind alle abhängig vom Datenmodell eines Services, daher basiert die PDL auf UML/P Klassendiagrammen [Rum11, Rum12, Sch12] und erweitert diese um Privacy-Eigenschaften mithilfe von Stereotypen. Die Beschreibung von Datenmodell und Privacy in einem Modell alleine genommen ermöglicht zwar die Integration von Privacy in die Entwicklung, allerdings erleichtert erst die konstruktive Nutzung von PDL Modellen im Entwicklungsprozess die Arbeit der Entwickler. Daher werden die PDL Modelle verwendet, um die Privacy Policy und das Data Usage Monitoring eines Services automatisch abzuleiten.

Privacy Policy

Jeder Dienst, welcher Nutzerdaten verarbeitet muss dem Nutzer (zumindest nach deutschem Recht) offenlegen, zu welchem Zweck welche Daten verarbeitet werden und welche Subdienstleister an der Datenverarbeitung beteiligt sind. Diese Informationen werden in einer Privacy Policy zusammengefasst, die der Nutzer bestätigen muss. Die Privacy Policy setzt sich üblicherweise aus einem Service spezifischen Teil und einem allgemeinen Haftungsausschluss zusammen, die beide von der Rechtsabteilung eines Service Anbieters ausgearbeitet werden. Wie bereits in Abschnitt 9.2 beschrieben, sind die Service spezifischen Informationen oft sehr vage formuliert in der Privacy Policy, da Entwickler der Rechtsabteilung Informationen über Datenverarbeitung nicht präzise genug darstellen können und die Rechtsabteilung ihrerseits nicht die detaillierten Designdokumente bzw. die Implementierung als Informationsquelle nutzen kann. Dieses Dilemma wird durch die PDL gelöst, indem detaillierte Informationen über die Nutzung von Daten innerhalb eines Services vom Entwickler beschrieben werden, sodass daraus der Service spezifische Teil der Privacy Policy generiert werden kann. Somit unterstützt die PDL Entwickler dabei das Prinzip Privacy-by-Design umzusetzen und Anforderung A3 zu erfüllen.

Herkömmliche Privacy Policies lassen dem Nutzer nur eine Wahl: Akzeptiere die gesamte Policy und nutze den Service oder nutze den Service überhaupt nicht. Durch die generierte interaktive Privacy Policy wird eine detailliertere Wahl ermöglicht, bei der Nutzer entscheiden können, welche Teile eines Services sie nutzen möchten. Dementsprechend muss der Nutzer dem Service auch nur eine Teilmenge der Daten, die bei der Nutzung des gesamten Services notwendig wären bereitstellen. Somit ist der Nutzer in der Lage effektiv die Privacy Policy gemäß seiner spezifischen Wahrnehmung von Privatsphäre anzupassen.

Für die Erklärung der PDL wird der Anwendungsfall Mobilitätsunterstützung für Senioren aus dem Projekt IPACS verwendet, der bereits in Abschnitt 9.3 beschrieben wurde. Listing 9.1 zeigt ein beispielhaftes PDL Modell, dass vom Service Entwickler erstellt wurde und Abbildung 9.2 zeigt die daraus generierte Privacy Policy für den Service. Im PDL-Modell werden alle Benutzerdaten, die im Service verarbeitet werden als Attribute modelliert und der Entwickler

PDL

```
package de.rwth.se.mobility;
import java.util.List;

classdiagram MobilityService {

  class RoutingService {
    <<use="Find people near me to help me on my trip.",
      optional="User.location">>
    void addPotentialHelpers();

    <<use="Respect the shopping list when calculating the route.",
      optional="User.shoppingList">>
    void addShoppingList();

    <<use="Recommend items when at the store
           based on shopping list.",
      unused="If you do not want to get advertised shopping items,
              the service costs $1 per month.",
      optional="User.shoppingList">>
    void recommendAdditionalItems();

    <<use="Plan your current route.", mandatory="User.location">>
    void planRoute();

    <<use="Inform someone else about where I am and where
           I am heading to.",
      optional="RoutingService.currentRoute">>
    Route retrieveMe();

    <<condition="A family member is searching for me.">>
    boolean familiyIsSearchingForMe();

    <<description="Next places to visit">>
    Route currentRoute;
  }

  class User {
    Location location;
    List<String> inabilities;
    List<String> shoppingList;
  }
}
```

Listing 9.1: Dieses PDL-Modell modelliert den Anwendungsfall Mobilitätsunterstützung.

gibt über den Stereotype `use` an einer Methode an, wie diese Nutzerdaten durch den Services verarbeitet werden. Ein Beispiel für den Einsatz eines solchen `use` findet sich an der Methode `planRoute` in Zeile 22 von Listing 9.1. Auf diese Art lässt sich die Verwendung von Nutzerdaten detailliert und verständlich beschreiben, da der Entwickler für jede Methode genau benennen kann welche Operationen auf den Daten ausgeführt werden. Um Nutzerdaten in der Privacy Policy zu benennen wird ein für den Nutzer verständlicher Name benötigt. Hierzu wird der Name des Attributes in der PDL herangezogen, oder falls dieser rein technischer Natur ist, beispielsweise wenn ein gegebenes Interface implementiert werden muss, wird der Stereotyp `description` verwendet um das Attribut für den Nutzer zu beschreiben. Dies wird in Zeile 33 von Listing 9.1 genutzt, um die Variable `currentRoute` zu beschreiben.

Um dem Nutzer eine Anpassung der Servicefunktionalität an seine individuellen Privacy-Bedürfnisse zu ermöglichen kann der Service Entwickler Parameter einzelner Methoden, also die Nutzerdaten, die die Methode verarbeitet, mit Hilfe des entsprechenden Stereotyps als `optional` bzw. `mandatory` (erforderlich) deklarieren. Jeder als `optional` markierte Parameter stellt eine Wahlmöglichkeit für den Nutzer in der Privacy Policy des Services dar. Die Methode `addShoppingList` in Zeile 13 aus Listing 9.1 kann dementsprechend das Attribut `shoppingList` der Klasse `User` verwenden, oder ihre Funktion auch ohne Zugriff auf die Daten verrichten. Wenn die Funktionalität des Services sich ändert, sobald der Zugriff auf optionale Parameter nicht möglich ist, müssen die Auswirkungen im Stereotyp `unused` für den Nutzer erläutert werden. Zum Beispiel erhält der Nutzer für die Methode `recommendAdditionalItems` in Zeile 20 aus Listing 9.1 zwei Optionen:

1. Erlaube dem Service die Einkaufsliste zu benutzen, um zielgruppenspezifische Werbung anzuzeigen, wodurch der Betrieb des Services finanziert wird, oder

2. Verweigere dem Service den Zugriff auf die Daten, sodass der Service nicht über Werbung finanziert wird, sondern durch eine monatlich Gebühr von $1.

Somit muss der Nutzer für jeden `optional` Parameter zwischen den sich gegenseitig ausschließenden Optionen `use` und `unused` entscheiden.

Wie bereits in Abschnitt 9.1 beschrieben, ändern sich die Privacy-Anforderungen von Nutzern, wenn sie sich beispielsweise plötzlich in einer Notsituation befinden. Da in solchen Situationen keine Zeitverzögerung durch eine zusätzliche Nutzerinteraktion geduldet wird, ist es nicht möglich dem Nutzer eine interaktive Privacy Policy vorzulegen, wenn die Daten benötigt werden, beispielsweise im Fall eines Unfalls. Stattdessen entscheidet der Nutzer bereits bei der Einrichtung eines (Notfall-)Dienstes über die Wahlmöglichkeiten, die die Privacy Policy bietet.

Für jeden Notfall, den ein Service erkennt bzw. der durch einen vertrauenswürdigen Dritten dem Service signalisiert wird gibt es eine Methode in der Service Implementierung, welche auf das Ereignis reagiert. Um einem Nutzer die Entscheidung zu ermöglichen, bei welchen Ereignissen ein Notfall vorliegt und seine Privacy Policy variiert werden soll, bietet die PDL den Stereotype `condition`. Dieser markiert Methoden, die Ereignisse anzeigen, welche der Nutzer in der Privacy Policy verwenden kann und gibt eine Beschreibung des Ereignises für den Nutzer an. Ein Beispiel für eine solche `condition` ist die Methode `familiyIsSearchingForMe` in Zeile 30 von Listing 9.1. Für jede solche `condition` erlaubt die generierte Privacy Policy dem Nutzer anzugeben, welche optionalen Daten dem Service zusätzlich bereitgestellt werden sollen,

wenn eine bestimmte Bedingungen eintritt. Die Auswertung der Bedingungen und somit auch die Entscheidung, ob ein Notfall vorliegt kann durch einen vertrauenswürdigen Dritten oder durch einen Trustpoint des Nutzers durchgeführt werden [HHK+14]. In beiden Fällen erhält der Service erst mit der Entscheidung, dass ein Notfall vorliegt, über die kryptographische Zugriffskontrolle Zugriff auf die Daten. Wenn die `condition` Methode keine Nutzerdaten benötigt, um einen Notfall festzustellen, ist eine Ausführung in der Cloud zuverlässiger, da bei einem Ausfall des Trustpoints oder der Verbindung zum Trustpoint der Trustpoint nicht die Autorisierung mittels kryptographischer Zugriffskontrolle durchführen kann. In der Privacy Policy erhält der Nutzer die Möglichkeit zu entscheiden, ob er darauf vertraut, dass ein Dritter seine Daten wirklich nur in einem Notfall freigibt oder, ob diese Aufgabe sein Trustpoint übernehmen soll. Somit kann der Nutzer für jede Bedingung entscheiden, ob ihm die Verfügbarkeit des Notfallservice oder die Aufrechterhaltung seiner Privatsphäre wichtiger ist.

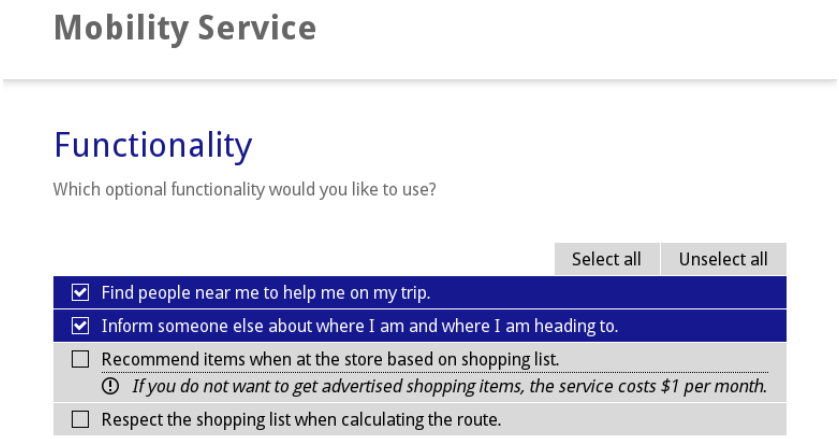
Data Usage Monitoring und Auditierung

Mittels interaktiver Privacy Policy spezifizieren die Nutzer, wie ihre Daten durch einen Service verwendet werden *sollen*. Um dem Nutzer transparent zu machen, wer *tatsächlich* auf seine Daten zugegriffen hat wird eine Datenbankabstraktionsschicht eingesetzt, die jeden Zugriff eines Services auf die Nutzerdaten protokolliert. Da das Datenmodell des Services als PDL Modell vorliegt, wird diese Schicht aus dem Modell generiert. Dabei müssen nicht alle Zugriffe auf Daten protokolliert werden, sondern nur solche, die als `optional` oder `mandatory` gekennzeichnet sind, da nur diese Nutzerdaten enthalten. Das Protokoll kann der Nutzer jederzeit einsehen und somit eine präzise Aussage darüber erhalten, welche Methode wann auf seine Daten zugegriffen hat. Damit der Nutzer bewerten kann ob diese Zugriffe seinen Erwartungen und der gewählten Privacy Configuration entsprechen, wird analog zur Privacy Policy der `use` Stereotyp verwendet, um jedem Methodenzugriff eine für den Nutzer verständliche Beschreibung zu geben.

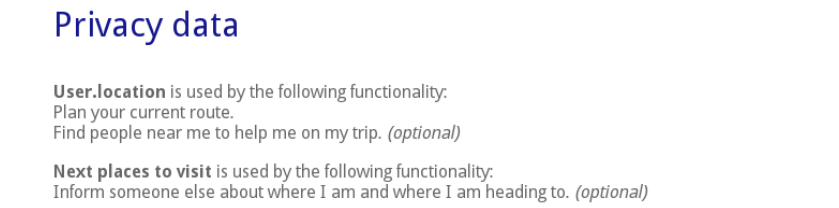
Dieses Protokoll über Zugriffe auf seine Daten ist für den Nutzer nur dann von Nutzen, wenn alle Methoden die Daten entsprechend der Beschreibung im `use` Stereotyp verwenden. Selbst wenn der Quellcode der Methoden offengelegt wird, ist ein Nutzer i.d.R. nicht in der Lage dies nachzuvollziehen. Daher wird diese Aufgabe an einen vertrauenswürdigen Auditor übertragen, auf dessen Urteil sich der Nutzer verlassen kann. Da die Beschreibung der Datennutzung direkt mit der Methode verbunden ist, die das beschriebene Verhalten implementieren soll, kann ein Auditor sich auf genau diese Teile der Implementierung konzentrieren und überprüfen, ob die Nutzerdaten wie beschrieben verarbeitet werden. Analog überprüft der Auditor, ob die in `condition` an einer Methode beschriebene Bedingung tatsächlich in der Implementierung überprüft wird und nur danach auf die Daten zugegriffen wird.

9.5. Generierung der Privacy Policy

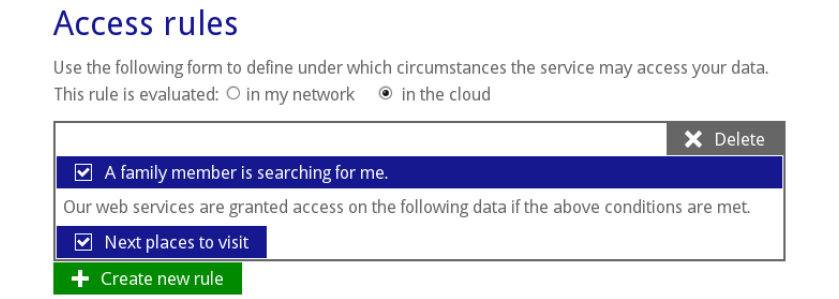
Für die Verarbeitung der PDL wurde auf Basis der existierenden MontiCore Infrastruktur ein Generator entwickelt, der die servicespezifische, interaktive Privacy Policy aus dem PDL Modell ableitet. Technologische Basis für die interaktive Privacy Policy bildet eine HTML Single-Page-Application, die jQuery [The15] für die Realisierung der Nutzerinteraktion verwendet. Abbildung



(a) Auswahl der zu nutzenden Service Funktionalität.



(b) Zusammenfassung der durch den Service genutzten Daten.



(c) Regeleditor, der die Definition von flexiblen Zugriffsrechten erlaubt.

Abbildung 9.2.: Screenshots der generierten, interaktiven Privacy Policy für den in Listing 9.1 definierten Anwendungsfall Mobilitätsunterstützung.

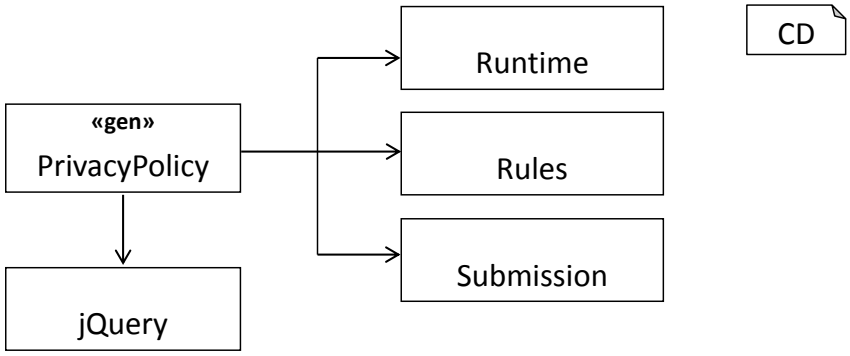


Abbildung 9.3.: Struktureller Zusammenhang in der Privacy Policy HTML Single-Page-Application.

9.3 zeigt die Komponenten dieser Anwendung. Jede generierte Privacy Policy verwendet drei JavaScript Komponenten: Die `Runtime` realisiert die allgemeine Nutzerinteraktion, `Rules` definiert das Verhalten des Regeleditors und `Submission` serialisiert die Informationen aus der Privacy Policy in ein Json Objekt, welches an den Cloud Service geschickt wird, wenn der Nutzer die Policy bestätigt.

Abbildung 9.2 zeigt die aus dem PDL Model aus Listing 9.1 abgeleitete Privacy Policy, deren Generierung im Folgenden beschrieben wird. Bei der Generierung der Privacy Policy wird für jede Methode in der PDL, die einen als `optional` markierten Parameter besitzt ein Eintrag im Abschnitt "Functionality" erstellt, der in Abbildung 9.2a dargestellt ist. Hier wird der `use`-Eintrag der Methode als Beschreibung der auswählbaren Funktionalität verwendet. Wenn ein `unused`-Eintrag vorhanden ist, wird dieser unterhalb der Beschreibung angezeigt, wenn die Funktionalität nicht ausgewählt ist. Listing 9.4 zeigt das FreeMarker Template, welches zur Generierung dieses Abschnitts der Privacy Policy verwendet wird. Dieses iteriert mit `#list` in Zeile 15 über die Methoden mit optionalen Parametern und erstellt für jede ein `li`-Element. Ein möglicherweise vorhandener `unused`-Eintrag wird in ein `p`-Element mit der CSS Klasse `unselected_info` übersetzt. Diese HTML-Elemente werden durch jQuery zur Ausführungszeit der Single-Page-Application in die in Abbildung 9.2 dargestellte Webseite übersetzt.

Im Abschnitt "Privacy data", der in Abbildung 9.2b dargestellt ist, wird angezeigt welche Daten bei der aktuellen Auswahl an Funktionalität durch welche Methoden des Services verarbeitet werden. Die Beziehungen zwischen Methoden und verarbeiteten Daten werden aus dem PDL Modell gelesen und in Programmcode übersetzt, der bei Auswahl einer Checkbox im Abschnitt "Functionality" für jedes genutzte Attribut den `use`-Eintrag anzeigt. Für Attribute, die als `optional` gekennzeichnet sind, wird ein Hinweis darauf an die `use`-Beschreibung angehängt. Außerdem wird bei Attributen, die eine `description` haben, wie beispielsweise `currentRoute`, diese Beschreibung statt ihres vollqualifizierten Namens angezeigt. Das in Listing 9.5 dargestellte FreeMarker Template übersetzt die Informationen aus der PDL wie zuvor beschrieben in HTML, welches durch jQuery zur Darstellung weiterverarbeitet wird.

Der Regeleditor im letzten Teil der interaktiven Privacy Policy, der in Abbildung 9.2c dargestellt ist, wird erstellt, sobald eine `condition` im PDL Modell vorhanden ist. Die Beschreibung

```
<article id="service_article">                                          HTML
  <header>
    <hgroup>
      <h2>Functionality</h2>
      <p>Which optional functionality would you like to use?</p>
    </hgroup>
  </header>

  <section id="service_section">
    <ul class="clickable clickable_standard_skin
        reverse_flow_panel separator_left">
      <li id="unselect_all">Unselect all</li>
      <li id="select_all">Select all</li>
    </ul>
    <ul id="services" class="clickable clickable_standard_skin
        checkable separator_top">
    [#list method_container.getOptionalMethods() as
        optional_method]
      <li class="service" id="${optional_method.getId()}">$
        {optional_method.getDescription()}
        [#if optional_method.hasUnusedInfo()]
          <p class="unselected_info">
            ${optional_method.getUnusedInfo()}
          </p>
        [/#if]
      </li>
    [/#list]
    </ul>
  </section>
</article>
```

Listing 9.4: Dieser Ausschnitt des FreeMarker Templates generiert den Abschnitt "Functionality" der HTML Single-Page-Application.

der `condition` wird dabei im ersten Teil des Regeleditors verwendet, um die möglichen Bedingungen aufzulisten, unter denen die im zweiten Teil des Regeleditors dargestellten Attribute freigegeben werden sollen. Durch das in Listing 9.6 dargestellte FreeMarker Template werden diese Informationen aus dem Modell in HTML übersetzt. In Zeile 9-13 werden die Bedingungen generiert und die Zeilen 19-29 generieren die freizugebenden Attribute.

Um eine einfache Integration in die HTML Beschreibung von Cloud Services zu ermöglichen, ist die interaktiven Privacy Policy als HTML Single-Page-Application realisiert. Um die generierte Privacy Policy in eine Servicebeschreibung zu integrieren erlauben Cascading Style Sheets (CSS) das Aussehen und Layout der Privacy Policy zu verändern ohne jedoch ihre Funktionalität zu beeinflussen. Nachdem der Nutzer die Privacy Policy seinen Bedürfnissen entsprechend angepasst hat wird diese durch klicken des Submit-Knopfes serialisiert und an den Cloud-Service gesendet. Danach obliegt es dem Service Provider diese Policy auch einzuhalten.

```
<article id="input_article">
  <header>
    <hgroup>
      <h2>Privacy data</h2>
    </hgroup>
  </header>
<section id="input_section">
[#if method_container.containsMandatoryMethod()]
  <ul>
    [#list attribute_container.getAttributes() as attribute]
    [#if attribute.isMandatory()]
      <li style="margin-top:1em">
        <p>
          <b>${attribute.getDescription()}</b>
          is used by the following functionality:
        </p>
        <ul>
          [#assign attribute_method_container = attribute.
             getMethodContainer()]
          [#list attribute_method_container.getMandatoryMethods()
             as mandatory_method]
            <li>${mandatory_method.getDescription()}</li>
          [/#list]
          [#list attribute_method_container.getOptionalMethods()
             as optional_method]
            <li class="service_info service_info_${optional_method
               .getId()}"></li>
          [/#list]
        </ul>
      </li>
    [/#if]
    [/#list]
  </ul>
[/#if]
```

HTML

Listing 9.5: Dieser Ausschnitt des FreeMarker Templates generiert den Abschnitt “Privacy data” der HTML Single-Page-Application.

9.6. Nutzerinteraktion

Aus der Perspektive des Service Nutzers dient die interaktive Privacy Policy als Fenster, um in die Verarbeitung seiner Daten durch den Services Einblick zu erhalten. Je nachdem welche Funktionalität der Nutzer auswählt, wird im Abschnitt “Privacy data” der Privacy Policy angezeigt, welche Daten zu welchem Zweck verarbeitet werden. Der Nutzer kann somit für jede Methode auswählen, ob er die damit verbundene Funktionalität nutzen möchte und sieht im Abschnitt “Privacy data” direkt das Resultat seiner Auswahl. Somit kann er seine Auswahl so anpassen, dass

```HTML
<ul class="rule">
  <li>
    <ul class="rule_header clickable clickable_red_skin
        delete_icon reverse_flow_panel">
      <li class="delete_rule">Delete</li>
    </ul>
  </li>
  <li>
    <ul class="rule_condition clickable clickable_standard_skin
        checkable separator_top separator_left separator_right">
      [#assign i = 0]
      [#list condition_container.getConditions() as condition]
        <li name="c${i}">${condition}</li>
        [#assign i = i + 1]
      [/#list]
    </ul>
  </li>
  <li>
    <p>Our web services are granted access on the following data
        if the above conditions are met.</p>
    <ul class="rule_data clickable clickable_standard_skin
        checkable flow_panel separator_bottom separator_left">
      [#list attribute_container.getAttributes() as attribute]
        [#if attribute.isOptional()]
          [#assign attribute_method_container = attribute.
              getMethodContainer()]
          <li name="${attribute.getId()}" class="invisible
            [#list attribute_method_container.getOptionalMethods()
                as optional_method]
              input_${optional_method.getId()}
            [/#list]">
              ${attribute.getDescription()}
          </li>
        [/#if]
      [/#list]
    </ul>
  </li>
</ul>
```

Listing 9.6: Dieser Ausschnitt des FreeMarker Templates generiert den Abschnitt “Access rules” der HTML Single-Page-Application.

sein Nutzen und die vom Service verwendete Daten seinen persönlichen Privacy-Anforderungen gerecht werden.

Der in Abbildung 9.2c dargestellte Regeleditors erlaubt es einem Nutzer mehrere Regeln angeben, die unabhängig voneinander ausgewertet werden. In jeder Regel gibt der Nutzer im oberen Teil des Editors Bedingungen an, die alle erfüllt sein müssen, damit die im unteren Bereich

ausgewählten Attribute freigegeben werden. Somit kann der Nutzer genau angeben welche seiner Daten unter welchen Bedingungen verarbeitet werden dürfen.

Durch den Einsatz der PDL werden detaillierte Informationen über die Verwendung von Nutzerdaten in Form der Privacy Policy bereitgestellt. Diese Privacy Policy kann von versierten Nutzern direkt zur Konfiguration eines Services genutzt werden, allerdings benötigen Privacy-Leihen eine einfachere Darstellung der Privacy Policy. Eine solche Darstellung können Privacy-Experten mit Hilfe der detaillierten Informationen aus der generierten Privacy Policy erstellen, indem sie Vorschläge für Privacy-Konfigurationen erarbeiten und diese für die entsprechende Zielgruppe anschaulich beschreiben. Somit erlaubt es die Privacy Policy den Experten mögliche Privacy-Konfigurationen, die ein Service bietet, zu bewerten und Empfehlungen auszusprechen.

Kapitel 10.

Evaluation und Fallstudien

In diesem Kapitel wird mittels Fallstudien und Evaluation gezeigt, wie die in dieser Arbeit entwickelten Ansätze und Werkzeuge im Rahmen von Softwareentwicklungsprojekten eingesetzt werden und wie das Ergebnis dieser Projekte sich durch den Einsatz verändert. Die verwendeten Szenarien stammen aus den Projekten SensorCloud [EHH+14] und Intelligent Privacy-Aware Cloud-Based Services (IPACS) [Pro15]. Zuerst wird eine Evaluation von MontiSecArc an einem Szenario aus dem SensorCloud Kontext durchgeführt. Anschließend wird gezeigt, wie MontiCoder auf HTML und JavaScript angewandt wird und zuletzt wird der Einsatz der PDL an einem Szenario aus dem Projekt IPACS demonstriert.

10.1. Anwendung von MontiSecArc im Kontext des Projekts SensorCloud

Ziel dieser Evaluation ist es, die Methodik mit der MontiSecArc angewandt wird, zu verdeutlichen und den Einfluss von MontiSecArc auf die Security-Architektur eines Systems zu zeigen. Dazu wird zunächst beschrieben, wie die Evaluierung durchgeführt wurde und im Anschluss die Methodik von MontiSecArc anhand einer exemplarischen Lösung der Aufgabenstellung in einer Fallstudie dargestellt. Die Betrachtung von MontiSecArc schließt mit der Auswertung der durchgeführten Evaluation, bei der Metriken auf die erstellten Modelle angewandt werden und eine STRIDE-Analyse durchgeführt wird.

10.1.1. Entwurf und Durchführung der Evaluation

Zielgruppe für MontiSecArc und die dazugehörigen Analysewerkzeuge sind Softwareentwickler. Daher wurden für die Evaluation im Rahmen des Praktikums Built it, Break it, Fix it durch eine Quotenauswahl [Die07, S. 390-399] aus 59 Master Studenten 14 ausgewählt, die bereits über Erfahrung in Softwareentwicklungsprojekten verfügten und ihr grundlegendes Wissen über Web-Security in einem einfachen Praxistest zeigen konnten. Diese wurden auf 9 Gruppen so aufgeteilt, dass jede Gruppe mindestens einen erfahrenen Softwareentwickler hat. 13 weitere Teilnehmer aus dem Bachelor Informatik Studiengang, die sich nach eigener Interessenlage für das Praktikum entschieden hatten, wurden gleichmäßig auf diese Gruppen verteilt. Eine Selektion bezüglich Vorwissen konnte bei den Bachelor Studenten nicht durchgeführt werden.

Um bezüglich des Gebiets der Software Security ausgewogene Gruppen zu bilden mussten alle Teilnehmer ein Thema aus den Bereichen Build, Break oder Fix auswählen, das sie in einem

Vortrag aufbereiten. Build-Themen waren dabei rein aus dem Gebiet der Software Entwicklung ohne Security-Hintergrund, Break-Themen stellten Angriffe und Angriffswerkzeuge für Webapplikationen vor und Fix-Themen befassten sich mit Schutzmaßnahmen für Webapplikationen. Um Dreiergruppen zu bilden wurde jeder Gruppe je ein Teilnehmer zugewiesen, der ein Build-, Break-, bzw. Fix-Thema bearbeitet hatte. Dabei wurde auf eine möglichst gleichmäßige Verteilung von Master und Bachelor Studenten auf die Gruppen geachtet. Die so geformten Gruppen sollten agilen Teams in ihrer Struktur ähneln, da es einen Lead-Entwickler gab und das Wissen über die verschiedenen Bereiche der Software Security möglichst gleichmäßig verteilt sein sollte.

Diese Gruppen erhielten dieselben Systemanforderungen, aus denen sie eine Security-Architektur erstellen sollten. Fünf Gruppen verwendeten MontiSecArc als Security-Architekturbeschreibungssprache zusammen mit einem Werkzeug, welches vordefinierte Analysen auf dem MontiSecArc-Modell durchführte. Die übrigen vier Gruppen dienten als Kontrollgruppe und verwendeten ausschließlich graphische Architekturmodelle ohne speziellen Fokus auf Security und ohne Werkzeugunterstützung zum Entwurf der Security-Architektur.

10.1.2. Szenario und exemplarische Lösung

Ziel des Projekts SensorCloud, welches vom Bundesministerium für Wirtschaft und Energie im Rahmen der Trusted Cloud Initiative gefördert wurde, war es die Entwicklung von Cyber Physical Systems durch die Nutzung von Cloud Technologien zu vereinfachen und durch die Anwendung von kryptographischer Zugriffskontrolle die Datensicherheit von in der Cloud gespeicherten Daten sicherzustellen, um die Nutzerakzeptanz von Cloud Systemen zu steigern. Das im Rahmen der Evaluation durch die Gruppen zu entwickelnde CPS ist ein Warenwirtschafts- und Kassensystem eines Supermarkts in Anlehnung an das Common Component Modeling Example (CoCoME) [HKW+08]. Im Vergleich zu CoCoME enthält das Szenario jedoch mehr Systeme und unterschiedliche Akteure, sodass Interessenskonflikte und somit Bedarf an Security Schutzmaßnahmen auftreten. Folgende informelle Anforderungen wurden an das System gestellt:

1. An einer **Selbstbedienungskasse** können Kunden ihre Waren mit Bargeld, per Kreditkarte oder online bezahlen. Außerdem ist es möglich Bargeld an der Kasse abzuheben.

2. Eine **Online Vorbestellung** kann durchgeführt werden, bei der auch bereits gezahlt werden kann, um die Schnellkasse benutzen zu können. Die Webserver, auf denen die Online Vorbestellung abgewickelt wird, stehen in einem Rechenzentrum.

3. Die Regale im Supermarkt sind mit **elektronischen Preisschildern** ausgerüstet, die den Preis der Waren anzeigen und auch die Waren eindeutig identifizieren.

4. Ein **Warenwirtschaftssystem** wird genutzt, um die Regale und das Hochregallager aufzufüllen. Die Buchhaltung benötigt in Echtzeit Zugang zu den Informationen über den Lagerbestand.

5. Ein Lieferant bringt Waren zum **Hochregallager** und von dort aus zum Supermarkt.

6. Lagerarbeiter können den **Lagerstand** korrigieren, wenn Waren schlecht werden.

Das MontiSecArc Analyse Werkzeug wurde zusammen mit einem initialen Beispiel über das Git Repository `https://git.rwth-aachen.de/monticore/MontiSecArc` bereitgestellt und benötigt zur Ausführung eine Java Laufzeitumgebung in Version 7. Im Verzeichnis `MontiSecArc/examples/montiSecArcAnalysis/` befindet sich in der Datei `startExample.sh` ein Bash Script zum Ausführen der Analysen auf dem MontiSecArc-Modell aus der Datei `examples/montiSecArcAnalysis/src/test/resources/de/rwth/se/supermarket/SupermarketTradingSystem.secarc`. Durch Anpassen des Scripts oder der Datei `SupermarketTradingSystem.secarc` kann dieses Beispiel zu einer Security-Architektur für den Supermarkt weiterentwickelt werden, welche die zuvor dargestellten Anforderungen erfüllt.

MSA

```
1 package de.rwth.se.supermarket;
2
3 component SupermarketTradingSystem {
4
5   trustlevel +0 "customers might be attackers";
6   accesscontrol off;
7   connect supermarket.sold -> highRackWarehouse.order;
8
9   component Supermarket supermarket{
10    port out Item sold;
11  }
12
13  component HighRackWarehouse highRackWarehouse {
14    trustlevel +2;
15    accesscontrol on;
16    access stockManager, storeManager, cashier;
17    port in Item order;
18  }
19 }
```

Listing 10.1: Erste unvollständige Version einer Security-Architektur für das Supermarkt Szenario.

Ein erster Entwurf der Security-Architektur des Supermarkt Szenarios in dem nur ein kleiner Teil der Anforderungen umgesetzt ist wird in Listing 10.1 dargestellt. Bereits diese frühe und unvollständige Security-Architektur kann der Entwickler automatisiert auf Flaws untersuchen. Die Ausgabe dieser Analyse ist in Listing 10.2 zu sehen und zeigt einen Fehler in Zeile 13 des Modells in der Komponente `HighRackWarehouse` an. Hier wurde der in Abschnitt 5.3.1 beschriebene Flaw erkannt, dass die Komponente `HighRackWarehouse` zwar über Rollen verfügt, aber kein `identity` Link definiert wurde, sodass nicht klar ist wie Nutzer der Komponente authentisiert werden. Der Entwickler steht also jetzt vor der Entscheidung, ob ein zentrales Identity Management System angeschlossen werden soll, oder eine direkte Authentication zwischen den Komponenten erfolgen soll. Die weiteren als Warnung markierten Ergebnisse der Analyse sind Flaw Smells (siehe Abschnitt 5.3), die auf mögliche Flaws hindeuten bzw. einem Entwickler oder Auditor die Suche nach Flaws vereinfachen sollen. So wird beispielsweise ab

Zeile 9 der Analyse zusammengefasst, welche Zugriffsrechte im Modell spezifiziert wurden. Dies ist besonders bei größeren Modellen hilfreich, um einen besseren Überblick über die vergebenen Rechte im Gesamtsystem zu erhalten.

Durch eine Integration dieser Analyse in den Buildprozess werden Flaws bereits während der Entwicklung identifiziert und dem Entwickler gemeldet. Wird ein Review des Systems durch einen unabhängigen Auditor durchgeführt, kann dieser das MontiSecArc-Modell und die Ergebnisse der Analyse nutzen, um schnell eine unabhängige Einschätzung über Weaknesses in der Security-Architektur zu erhalten.

10.1.3. Einfluss von MontiSecArc auf die erstellten Architekturbeschreibungen

Um den Einfluss der MontiSecArc Sprache und die Nutzung des Analysewerkzeugs auf die Qualität der Security-Architektur zu bestimmen werden die erstellten Architekturbeschreibungen bewertet. Allerdings ist es grundsätzlich schwierig eine zuverlässige Metrik aufzustellen, die alle Security-Eigenschaften zusammenfasst und bewertet [PC10]. Die Attack Surface Metrik [MW11] beispielsweise adressiert eher Probleme aus dem Bereich Language Security und zielt somit darauf ab Protokolle einzusetzen, die weniger Code enthalten, der durch den Empfänger ausgeführt werden muss. Auf eine Architektur angewandt fördert die Attack Surface Metrik den Einsatz des Single Access Point Security Pattern [GG12].

Daher wird neben einem quantitativen Vergleich der angewandten Metriken in Tabelle 10.3 auch die STRIDE Methode eingesetzt, um Flaws in der Architekturbeschreibung zu identifizieren. Um die STRIDE Methode möglichst objektiv durchzuführen, werden alle Architekturmodelle in Data Flow Diagrams übersetzt, auf denen mit dem Threat Modeling Tool von Microsoft [Mic15] eine automatisierte STRIDE-Analyse durchgeführt wird. Die Anzahl der dabei identifizierten Flaws sind in Tabelle 10.4 dargestellt.

Für alle hier präsentierten Ergebnisse, die eine Durchschnittsbildung verwenden, wird ein Wilcoxon-Mann-Whitney U-Test mit der Funktion `wilcox.test` in R [R C14] durchgeführt und der W-Wert in der Tabelle angegeben. Mit diesem Test wird überprüft, ob die erhobenen Daten genügend Signifikanz aufweisen, damit die Nullhypothese abgelehnt werden kann und stattdessen die überprüfte Hypothese gilt. Der in der Tabelle dargestellte p-Wert gibt die Wahrscheinlichkeit für die Nullhypothese an. Signifikante Ergebnisse, bei denen $p < 0{,}05$ gilt werden mit einem x gekennzeichnet und Ergebnisse bei denen $p < 0{,}1$ gilt, welche annähernd signifikant sind, werden mit einem o gekennzeichnet.

In Tabelle 10.3 wird die Komplexität der erstellten Architekturmodelle durch die Anzahl der Komponenten und Konnektoren, sowie die zyklomatische Komplexität nach McCabe [McC76] dargestellt. Die mit MontiSecArc erstellten Modelle enthielten dabei durchschnittlich mehr Komponenten und Konnektoren und auch die zyklomatische Komplexität ist im Durchschnitt bei MontiSecArc-Modellen höher. Eine möglich Erklärung für diesen Anstieg der Komplexität ist die Verwendung einer textuellen Notation, die es erlaubt Modelle zu erstellen, die sich über mehrere Seiten erstrecken, was bei graphischen Modellen eher selten vorkommt. Dieser erhöhte Detailgrad in der Spezifikation von MontiSecArc-Modellen beeinflusst unter Umständen die Ergebnisse

MSAA

```
1 Start analysis
2
3 tutorial.secarc - parsed successfully!
4 tutorial.secarc:13:2 - ERROR: For the component de.rwth.se.
      supermarket.SupermarketTradingSystem.HighRackWarehouse are
      roles defined. These roles must be authenticated by an identity
       link.
5 tutorial.secarc:3:1 - WARNING: The component
      SupermarketTradingSystem has the following roles: stockManager,
       storeManager, cashier
6 tutorial.secarc:7:2 - WARNING: Taint tracking: The trustlevel
      becomes higher with the transition supermarket.sold ->
      highRackWarehouse.order. Before the input is used, it must be
      filtered. The following pass is used: 'supermarket.sold ->
      highRackWarehouse.order'. The filter is marked with a *. If the
       filter is missing, the input cannot be used.
7 tutorial.secarc:7:2 - WARNING: The unencrypted connector
      supermarket.sold -> highRackWarehouse.order is embedded in a
      component with a low trustlevel: +0.
8 tutorial.secarc:9:2 - WARNING: The component de.rwth.se.
      supermarket.SupermarketTradingSystem.Supermarket has the
      derived trustlevel +0
9 tutorial.secarc:13:2 - WARNING: The component HighRackWarehouse
      has the following roles: stockManager, storeManager, cashier
10 tutorial.secarc:16:3 - WARNING: The role stockManager has access
      to the following ports and components: de.rwth.se.supermarket.
      SupermarketTradingSystem.HighRackWarehouse
11 tutorial.secarc:16:3 - WARNING: The role storeManager has access
      to the following ports and components: de.rwth.se.supermarket.
      SupermarketTradingSystem.HighRackWarehouse
12 tutorial.secarc:16:3 - WARNING: The role cashier has access to the
       following ports and components: de.rwth.se.supermarket.
      SupermarketTradingSystem.HighRackWarehouse
13 tutorial.secarc:17:8 - WARNING: The port order has the following
      roles: stockManager, storeManager, cashier
14 WARNING: Regarding the configuration the result of the metrik is:
      9
15
16 Done
```

Listing 10.2: Ausgabe der Security-Architekturanalyse.

der im folgenden verwendeten Metriken. Ein signifikanter Unterschied in der Komplexität der MontiSecArc-Modelle lässt sich jedoch nicht zeigen. Die Ergebnisse für die Attack Surface Metrik und die Anzahl der Trust Level ist jedoch signifikant. Daher lässt sich schließen, dass MontiSecArc-Modelle einer Architektur einen größeren Attack Surface besitzen, als Modelle, die nicht mit MontiSecArc erstellt wurden. Die Attack Surface Metrik bezieht dabei jedoch nicht die in MontiSecArc definierten Schutzmaßnahmen mit ein, sondern betrachtet nur die Ports der obersten Komponente und die darin verwendeten Datentypen. Da in MontiSecArc jeder Port einen Typ besitzt, geben Entwickler hier auch mehr Typen an, als in grafischen Modellen, die keine typisierten Ports fordern. Durch eine strengere Typisierung von Nachrichten, welche auch in der Implementierung geprüft wird, werden Input Validation Vulnerabilities jedoch verhindert [BDL+14], sodass MontiSecArc bereits mehr Implementierungsdetails für die Input Validation festlegt, welches jedoch zu einem größeren Wert in der Attack Surface Metrik führt.

Das zweite signifikante Ergebnis zeigt, dass Entwickler bei der Nutzung von MontiSecArc während des Entwurfs ca. 1,5 mal mehr Trust Level definieren als ohne MontiSecArc und somit eine Architektur entsteht, die das Designprinzip Compartmentalization stärker berücksichtigt. Dabei ist zu bemerken, dass keins der graphischen Modelle explizit eine Compartmentalization durchführt, sondern für die STRIDE Analyse die Komponenten auf oberster Ebene als Trust Level angenommen wurden, wenn aus der Bezeichnung der Komponenten klar wurde, dass diese als Systemgrenzen zu verstehen sind. Ohne diese Annahme würde das Ergebnis noch deutlicher ausfallen.

Um einen genaueren Eindruck in die Wirkung von MontiSecArc auf die Security-Architektur eines Systems zu erhalten, als dies durch die Attack Surface Metrik möglich ist, werden die Anzahl der mit STRIDE identifizierten Flaws in Tabelle 10.4 dargestellt. Diese werden nach den Angriffskategorien Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service und Elevation of Privilege aufgeteilt dargestellt und die Flaws aus der Kategorie Elevation of Privileges werden weiterhin in die einzelnen Angriffe Remote Code Execution, Impersonation, Changing the Execution Flow und Cross Site Request Forgery aufgeteilt, da in dieser Kategorie besonders viele Flaws identifiziert wurden. In der Kategorie Elevation of Privileges identifiziert die STRIDE-Analyse mit Remote Code Execution und Cross Site Request Forgery Weaknesses, die gar nicht in der Architektur vorhanden sein können, da diese erst in der Implementierung entstehen. Die von der STRIDE-Analyse gemeldeten Flaw Zahlen sind also auch mit Falsch-Positiv Ergebnissen behaftet, bzw. identifizieren Weaknesses, die erst noch entstehen könnten und die ein Auditor bei einer manuellen Überprüfung achten sollte. Vergleicht man die Anzahl der Flaws in MontiSecArc-Modellen mit der Kontrollgruppe, bedeutet ein Unterschied, dass weniger Flaws in einer der beiden Gruppen *spezifiziert* wurden. Da jedoch, wie in Tabelle 10.3 zu sehen ist, MontiSecArc-Modelle detaillierter sind und auch mehr Trust Level spezifizieren, kann dies dazu führen, dass auch mehr Flaws in den MontiSecArc-Modellen gefunden werden. Insbesondere die Anzahl der Trust Level beeinflusst wie viele Flaws die STRIDE-Analyse findet, da in einem DFD ohne eine Trust Boundary ausschließlich Impersination Flaws identifiziert werden und alle anderen Flaws erst durch hinzufügen einer Trust Boundary erkannt werden. Somit lässt sich für die Kategorien Denial of Service, Elevation of Privileges und Repudiation in denen mehr Flaws in MontiSecArc-Modellen identifiziert wurden nicht schließen, dass der Einsatz von MontiSecArc zu mehr Flaws in der Architektur führen.

Tabelle 10.3.: Kennzahlen der erstellten Architekturbeschreibungen

Werkzeug	**Architekturkomponenten**	**Architekturkonnektoren**	**McCabe**	**Attack Surface**	**Trust Level**
ohne	8	16	10	5	4
ohne	6	6	5	13	4
ohne	10	10	10	28	3
ohne	11	21	15	14	3
MontiSecArc	9	17	23	43	5
MontiSecArc	11	18	19	30	6
MontiSecArc	12	14	8	18	7
MontiSecArc	13	25	24	31	4
MontiSecArc	8	14	33	62	5
∅ **ohne**	8,75	13,25	10	15	3,5
∅ **MontiSecArc**	10,6	17,6	21,4	36,8	5,4
W	5	6	3	1	1
p	0,2663	0,3893	0,1099	0,03175	0,03267
Signifikanz	-	-	-	x	x

Für die Kategorien Information Disclosure, Spoofing und Tampering identifiziert die STRIDE-Analyse trotz der zuvor beschriebenen ungünstigen Voraussetzungen im Durchschnitt weniger Flaws in den MontiSecArc-Modellen als in der Kontrollgruppe. Da jedoch die Signifikanz zu gering ist, kann die Nullhypothese nicht ausgeschlossen werden, sodass für eine wirklich belastbare Aussage eine größere Evaluation mit mehr Teilnehmern notwendig ist.

Tabelle 10.4.: Anzahl der in der STRIDE Analyse identifizierten Weaknesses

Werkzeug	**Denial of Service**	**Elevation of Privileges**	Remote Code Execution	Impersonation	Changing the Execution Flow	Cross Site Request Forgery	**Information Disclosure**	**Repudiation**	**Spoofing**	**Tempering**	**Summe**
ohne	22	46	10	16	10	10	10	12	22	10	158
ohne	12	24	6	6	6	6	4	6	8	4	76
ohne	14	30	7	9	7	7	7	7	14	7	102
ohne	12	39	6	21	6	6	6	6	12	8	116
MontiSecArc	26	50	13	17	13	7	0	13	6	0	132
MontiSecArc	26	58	13	19	13	13	9	13	12	14	177
MontiSecArc	20	44	10	14	10	10	0	10	4	0	112
MontiSecArc	26	64	13	25	13	13	7	13	18	8	187
MontiSecArc	20	44	10	13	10	10	0	10	0	2	109
∅ **ohne**	15	34,75	7,25	13	7,25	7,25	6,75	7,75	14	7,25	113
∅ **MontiSecArc**	23,6	52	11,8	17,6	11,8	10,6	3,2	11,8	8	4,8	143,4
W	2	2	1	6	1	2,5	14,5	2	15,5	13,5	5
p	0,05945	0,06506	0,0304	0,4127	0,0304	0,07724	0,3169	0,05945	0,2187	0,4587	0,2857
Signifikanz	o	o	x	-	x	o	-	o	-	-	-

Der Zeitaufwand für die Erstellung der Security-Architekturbeschreibungen wurde per anonymisiertem Fragebogen von den Teilnehmern abgefragt. Dieser lag für MontiSecArc bei durchschnittlich 25 Stunden und für die Kontrollgruppe bei 9 Stunden und 47 Minuten mit einem W-Wert von 3166 und einem p-Wert von 0,01238 ist dieses Ergebnis signifikant. Für dieses Ergebnis gibt es mehrere Einflussfaktoren. Zum einen ist in diesen Werten auch die Einarbeitungszeit enthalten, welche die Teilnehmer für die Verwendung der MontiSecArc Sprache und das Analysewerkzeug aufbringen mussten. Wohingegen die Teilnehmer in der Kontrollgruppe eine ihnen bekannte Sprache ohne Werkzeugunterstützung verwendeten. Zum anderen weisen die erstellten MontiSecArc-Modelle im Vergleich zur Kontrollgruppe im Durchschnitt die doppelte inhaltliche Komplexität auf. Es wurden also neben Security Eigenschaften auch mehr inhaltliche Details modelliert.

Wie bereits dargestellt führt der Einsatz von MontiSecArc dazu, dass mehr Trust Level und Schutzmaßnahmen modelliert werden. Dies ist eine wesentliche Voraussetzung für eine erfolgreiche automatische bzw. effiziente manuelle Security Architektur Analyse. Um diese Analyse zu ermöglichen und somit Flaws frühzeitig erkennen und beheben zu können ist es daher notwendig, mehr Zeit auf die Modellierung zu verwenden.

10.2. Anwendung von MontiCoder für HTML und JavaScript

Um zu zeigen, dass der in Kapitel 8 beschriebene Ansatz zur Generierung von kontextsensitiven Encodern mit dem Werkzeug MontiCoder für Sprachen funktioniert, die in der Praxis im Einsatz sind, wurde eine Fallstudie mit HTML und JavaScript durchgeführt. In diesem Szenario ist Cross-Site-Scripting ein weitverbreitetes Problem, welches durch die Nutzung des generierten kontextsensitiven Encoder verhindert werden soll. Um dies zu überprüfen stehen verschiedene Testwerkzeuge zur Verfügung, mit denen Webapplikationen auf XSS Vulnerabilities überprüft werden können.

Für die beiden Sprachen HTML und JavaScript wurde je eine Grammatik mit Encoding Tables definiert und diese wie in Abschnitt 8.3.1 beschrieben zu einer Sprache kombiniert. Durch die unterschiedlichen Kontexte in denen JavaScript in HTML auf verschiedene Weise eingebettet wird, erhöht sich die Komplexität der HTML Grammatik im Vergleich zu einer HTML Grammatik, die lediglich zum Erkennen von HTML geeignet ist. Die Zeichen <, > und `&` müssen in jedem Body eines HTML-Tag encoded werden, außer im `Script`-Tag, wo sie als Teil von JavaScript genutzt werden und der Browser daher in diesem Kontext die Zeichen *<*, *>* und *&* nicht dekodiert. Daher wurden in der HTML Grammatik zwei unterschiedliche Token für den Body eines Tags definiert. Einer für den `Script`-Tag und einer für alle anderen HTML-Tags.

Im Gegensatz dazu werden die Zeichen <, >, `&` und `"` in Kontexten, wo HTML und JavaScript erlaubt sind (wie beispielsweise in Tag-Attributen), genauso encodiert, wie in allen anderen Kontexten, wo nur HTML erlaubt ist. Diese unterschiedliche Encodierung führt dazu, dass der Befehl `alert('&quot;');` ein Fester mit einem Anführungszeichen öffnet, wenn er in einem `onclick`-Attribut verwendet wird, und der gleiche Befehl in einem `Script`-Tag ein Fenster mit dem Inhalt `&quot;` öffnet. Genauso funktioniert der Vergleich `(5 &lt; 6)` in einem `onclick`-Attribut, aber nicht in einem `Script`-Tag. Diese Spezialfälle ließen sich mit Hilfe der Encoding Tables implementieren, da jeder unterschiedliche Fall des Encoding in einem

bestimmten Token gilt, dass zum Teil über seinen Namen identifiziert werden musste.

MontiCoder ist eine Erweiterung von MontiCore, sodass die Entwicklung von Sprachen und Generierung von Sprachverarbeitungswerkzeugen genauso wie in MontiCore funktioniert [Kra10, Sch12]. Auch die Struktur des MontiCoder Git Repositories `https://sselab.de/lab9/private/git/MontiCoder/` ist an die von MontiCore angelehnt. So sind unter `core/mclang` die zuvor beschriebenen Sprachen zu finden, aus denen MontiCoder Parser, Decoder, Encoder und Unparser generiert. Diese generierten Werkzeuge bilden zusammen mit der Sprache ein Maven Artefakt, welches in einer beispielhaften Webapplikation, die unter `example/SimpleEncoderWebApp` im Repository liegt, genutzt wird. Durch den Einsatz des Maven Buildsystems [Apa15a] lassen sich die Sprachen und die Webapplikation unabhängig voneinander bauen, sodass nach einem Aufruf von `mvn install` im Verzeichnis `core`, MontiCoder und die Maven Artefakte, welche die Sprachen enthalten, gebaut werden.

Die Webapplikation wird ebenfalls durch einen Aufruf von `mvn install`, jedoch im Verzeichnis `example/SimpleEncoderWebApp` gebaut und mit `mvn jetty:run` unter der Adresse `http://localhost:8080/ex01` gestartet. Sie verwendet das Template aus Listing 10.5, welches im Repository unter `src/main/webapp/WEB-INF/templates/xhtmltemplate.xhtml` zu finden ist, um eine Webseite zu generieren. Die Webseite verwendet absichtlich keine JavaScript Funktionen, die den DOM der Webseite nach dem Parsen im Browser verändern. Somit wird das in Abschnitt 8.4.2 beschriebene Problem vermieden, dass das Encoding von Eingaben, die in solchen Funktionen verwendet werden, nicht alleine durch den Webserver durchgeführt werden kann.

Das Template wird durch die Webapplikation wie in Abschnitt 8.5 beschrieben zuerst geparsed und zur Initiierung des AST des Ausgabedokuments, also der generierten Webseite, verwendet. Im Anschluss werden die mit `#...#` im Template markierten Platzhalter, die in einzelne Knoten des AST geparsed wurden ersetzt. Dabei wird der Platzhalter `#name#`, welcher in verschiedenen Kontexten von HTML und JavaScript verwendet wird, mit Eingabedaten ersetzt, die die Webapplikation über den URL Parameter `name` erhält und `#actionURL#` wird durch die URL der Webapplikation ersetzt. Für die weitere Verarbeitung des AST durch den Unparser spielt es keine Rolle, dass der Wert für den Parameter `#actionURL#` nicht vom Nutzer der Webapplikation beeinflusst werden kann, sondern durch die Programmlogik festgelegt wird. Der aus der Grammatik generierte Unparser wendet für alle Knoten des AST das für den jeweiligen Token in der Grammatik definierte Encoding an und stellt so sicher, dass beim Unparsen des AST in einen String keine Zeichenketten ausgegeben werden, die Control-Token der Sprache darstellen. Der so erzeugte String wird als Webseite an den Client der Webapplikation ausgeliefert.

Diese Webapplikation wurde im Rahmen der Fallstudie auf Cross-Site-Scripting Vulnerabilities getestet mit den in der Praxis üblichen automatisierten und manuellen Penetrationstests, um den generierten Encoder zu testen. Für den automatisierten Test wurde der Zed Attack Proxy (ZAP) [Ope15d] verwendet, welcher als Interception Proxy zwischen Browser und Webapplikation läuft und die Anfragen des Browsers, die bei der legitimen Nutzung der Webapplikation entstehen als Vorlage für Anfragen mit bösartigen Eingabewerten nutzt.

Im ersten Schritt wurde ein “Active Scan” verwendet um XSS Vulnerabilities automatisiert zu identifizieren. Für diesen Test wurden die Optionen “Cross Site Scripting (Refelected)”, “CRLF injection” und “Parameter tampering” mit dem Schwellwert “Low” für Benachrichtigungen über

```HTML
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Example Page</title>
    <meta name="viewport" content="width=device-width, initial-
        scale=1.0"/>
  </head>
  <body>
    <div>
      <h1>User Registration Formular</h1>
      <p>Hello <span>#name#</span></p>
      <form method="GET" action="#actionURL#">
        <div>
        <label for="input">Name: </label>
        <input type="text" id="input" name="name" value="#name#"/>
        </div>
        <div>
          <input type="submit" value="Register" />
        </div>
      </form>
      <a href="javascript:callScript();">Login</a>
      <div>
        <p>#name#</p>
        <button onclick="alert(&quot;#name#&quot;);">Test</button>
        <button onclick="alert('#name#');">Test</button>
        <input type="text" name="input" value="#name#"/>
        <script>
          var name ="#name#" + '#name#';
        </script>
      </div>
    </div>
  </body>
</html>
```

Listing 10.5: Eine XHTML Seite in die an den mit #...# gekennzeichneten Stellen Eingabedaten eingesetzt werden.

```HTML
...
<button onclick="alert(&quot;;alert(1)&quot;);">Test</button>
...
<button onclick="alert(&apos;;alert(1)&apos;);">Test</button>
...
```

Listing 10.6: Durch ZAP als XSS erkannte Antworten der Webapplikation.

mögliche Schwachstellen konfiguriert und die Intensität dieser Tests wurde auf “Insane” gesetzt. Mit dieser Konfiguration sendete ZAP 100 Anfragen mit bösartigen Eingabewerten für den URL Parameter `name` und erkannte anhand der Antworten zwei Eingaben als erfolgreiches XSS. Als Eingabe verwendete ZAP `;alert(1)` und die als XSS erkannten Ausgaben der Webapplikation sind in Listing 10.6 dargestellt.

In beiden Fällen handelt es sich um Fehlalarme, da beide Werte korrekt in den JavaScript String encoded wurden und der durch ZAP beabsichtigte JavaScript Befehl `alert` nicht ausgeführt wird. Dies deutet vielmehr darauf hin, dass ZAP den im `onclick`-Attribut bereits im Template vorhandenen `alert` Befehl als den durch die Eingabe injecteten erkennt.

Im zweiten Schritt wurden die XSS Angriffs-Strings aus FuzzDB, die in ZAP integriert sind verwendet um XSS in der Applikation zu identifizieren. Diese Angriffe werden zwar durch ZAP automatisiert ausgeführt, über den Erfolg muss jedoch manuell entschieden werden. Daher wurden die Angriffe mit ähnlichem Muster in Gruppen zusammengefasst und aus jeder Gruppe mehrere Angriffe ausgewählt, sodass schlussendlich elf Angriffs-Strings verwendet wurden. Diese Strings wurden in allen Kontexten, in denen der Platzhalter `#name#` im Listing 8.6 verwendet wird, getestet und manuell im Browser überprüft, ob der XSS Angriff erfolgreich war. Alle getesteten Strings wurden korrekt encoded, sodass der Browser sie anzeigte und nicht als JavaScript ausführte.

10.3. Anwendung der Privacy Development Language im Projekt IPACS

In dieser Fallstudie soll die Methodik mit der die PDL bei der Entwicklung von Cloud Services angewandt wird, um die Privacy des späteren Nutzers zu verbessern, verdeutlicht werden. Dazu wird ein Anwendungsfall aus dem Projekt Intelligent Privacy-Aware Cloud-Based Services (IPACS) [Pro15] verwendet. Im Projekt IPACS werden die Auswirkungen von Cyber Physical Systems auf die Privacy von Personen die sie nutzen in verschiedenen Bereichen des Lebens betrachtet. All diese Bereiche und die realisierten Anwendungsfälle werden im Kontext des demographischen Wandels betrachtet. Daher sind die typischen Anwender alleinstehende Rentnerinnen, die mit Hilfe der CPS ihren Alltag trotz ihres Alters selbstbestimmt und selbständig verleben können.

Im Rahmen dieser Fallstudie wird der Anwendungsfall Ambient Assisted Living (AAL) verwendet, bei dem die Wohnung einer Nutzerin mit Sensoren ausgestattet wird, die unterschiedliche Werte über den Gesundheitszustand der Bewohnerin messen und zur weiteren Verarbeitung an einen Cloud Service senden. Diese Sensoren sind unterschiedlich invasiv für die Privacy der Nutzerin. So kann z.B. durch eine Kamera oder einen Bewegungsmelder überprüft werden, ob sich jemand in der Wohnung bewegt. Ein Kamerabild enthält jedoch viele weitere Informationen, die ebenfalls durch den Cloud Service verarbeitet werden.

In diesem Szenario soll ein Cloud Service realisiert werden, der die Bewohner in einer Wohnung mithilfe einer Kamera überwacht und im Falle einer Notsituation entweder einen Angehörigen oder einen Arzt alarmiert. Wird von einem Arzt ein kritischer Gesundheitszustand festgestellt, so gibt das System außerdem den Zugriff auf die Krankenakten des Patientin für den Arzt frei. Zur Finanzierung des Cloud Services kann die Nutzerin entweder eine monatliche Gebühr an

den Service Provider entrichten, oder der Verarbeitung der Kamerabilder zur personalisierten Werbung zustimmen.

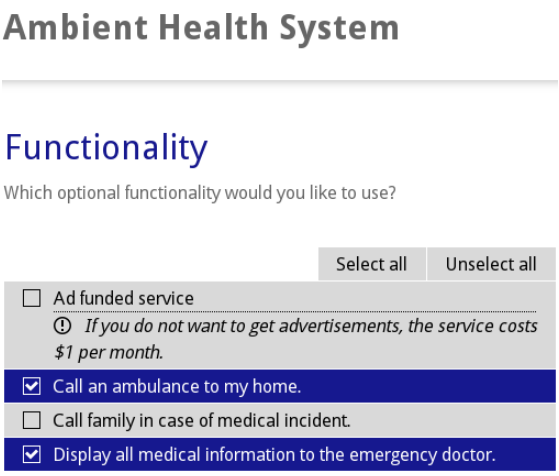
Um der Nutzerin Transparenz darüber zu geben, wie und zu welchem Zweck diese Kamerabilder verarbeitet werden und damit ihre Akzeptanz des Cloud Services zu erreichen, verwendet der Entwickler des AAL Cloud Services die PDL um das in Listing 10.7 dargestellte Datenmodell zu erstellen. Daraus erzeugt er eine interaktive Privacy Policy mit Hilfe des mit MontiCore und UML/P realisierten Generator, der im Repository unter `pri-generator` zu finden ist, und verwendet als Vorlage für die Integration des Generators in den Maven Build das Projekt `pri-service`, in dem das Modell aus Listing 10.7 unter `src/main/models/de/rwth/se/aal/example-aal.cd` angelegt ist. Dieses Datenmodell lässt sich zur Generierung von weiteren Teilen der Applikation verwenden, sodass der Entwickler weitere Generatoren mit Hilfe von MontiCore implementiert, die er ebenfalls über Maven in den Buildprozess integriert. Die aus dem Modell generierte Privacy Policy, die in Abbildung 10.8 dargestellt wird, ist mit HTML realisiert und lässt sich so in die Servicebeschreibung als zusätzliche Webseite integrieren. Diese Webseite bietet der Nutzerin eine Übersicht, wie die Daten aus ihrer Wohnung durch den Cloud Service genutzt werden (Abbildung 10.8b), um die von ihr erwünschten Dienstleistungen zu erbringen. Gleichzeitig ermöglicht die Privacy Policy es der Nutzerin auszuwählen, welche Teile des Services sie nutzen möchte (Abbildung 10.8a) und Daten nur unter bestimmten Bedingungen freizugeben (Abbildung 10.8c), sodass sie zwischen Nutzen des Services und ihrer Privatsphäre selbst entscheiden kann.

Diese Privacy Policy, und die Wahl die sie bietet, ist natürlich nur von Nutzen, wenn der Cloud Service die von der Nutzerin gewählte Policy auch wirklich umsetzt. Dies zu überprüfen ist zum einen eine technisch anspruchsvolle Aufgabe, da die Implementierung des Cloud Services auditiert werden muss, und zum anderen legen Cloud Services nur ungern ihre Implementierung offen, da dies ihren Wettbewerbsvorteil gegenüber Konkurrenten schmälern könnte. Daher überprüft ein unabhängiger Auditor, dass die interaktive Privacy Policy mit der Implementierung übereinstimmt, indem er die Methoden aus dem PDL-Modell in der Implementierung identifiziert und überprüft, ob die Angaben über die Nutzung der Daten in der PDL mit der Implementierung übereinstimmen. Außerdem prüft er, ob die Bedingungen zur Datenfreigabe wirklich in der Implementierung umgesetzt wurden. Um die Komplexität des Audits für den Nutzer komplett zu verbergen könnte ein Cloud Service Provider, der dem Nutzer eine Vielzahl von unterschiedlichen Services bereitstellt, nur solche in sein Angebot aufnehmen, die erfolgreich auditiert wurden. Durch diesen Prozess wird der Nutzerin versichert, dass ihre Daten nur in der von ihr in der Privacy Policy bestimmten und akzeptierten Weise verarbeitet werden.

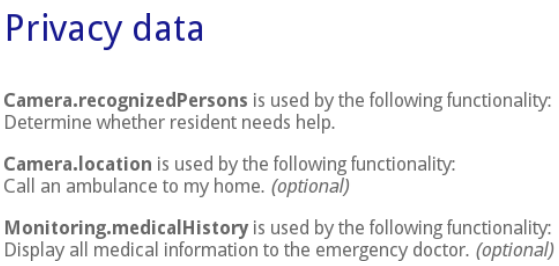
PDL

```
package de.rwth.se.aal;
import java.util.List;

classdiagram AmbientHealthSystem {

  class Camera {
    Room location;
    List<Person> recognizedPersons;
    VideoData stream;
  }

  class Monitoring {
    <<use="Determine whether resident needs help.",
      condition="Sensors detect that resident needs help.",
      mandatory="Camera.recognizedPersons">>
    boolean detectCriticalHealth();

    <<condition="Testified health professional detects that
                 resident needs help.">>
    boolean healthProfessionalDetectCriticalHealth();

    <<use="Ad funded service", unused="If you do not want to get
           advertisements, the service costs $1 per month",
      optional="Camera.recognizedPersons">>
    Ad getPersonalizedAd();

    List<MedicalCase> medicalHistory;
  }

  class NotificationManager {
    <<use="Call an ambulance to my home.",
      optional="Camera.location">>
    void placeEmergencyCall();

    <<use="Call family in case of medical incident.",
      optional="Camera.stream">>
    VideoData openVideoCall();

    <<use="Display all medical information
           to the emergency doctor.",
      optional="Monitoring.medicalHistory">>
    void provideMedicalHistory();
  }
}
```

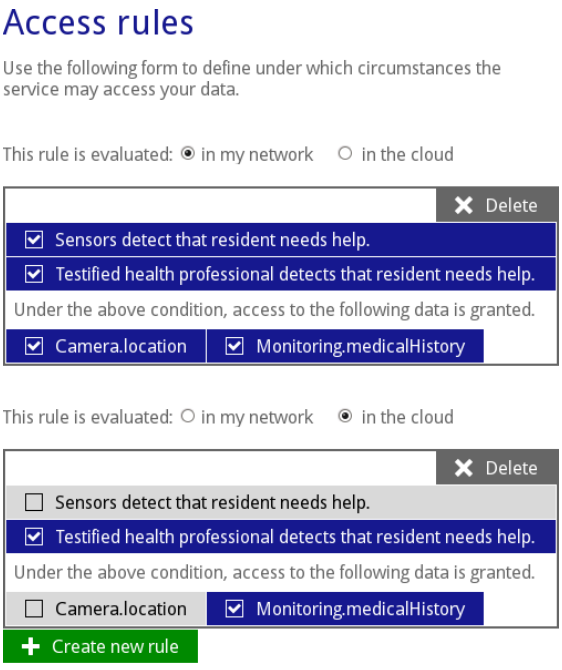
Listing 10.7: Umsetzung des Ambient Assisted Living Szenarios.



(a) Interaktive Auswahl der Funktionalität.



(b) Übersicht über die Nutzung der Daten.



(c) Der Regeleditor ermöglicht die Freigabe von Informationen unter bestimmten Bedingungen.

Abbildung 10.8.: Screenshots der automatisch generierten Privacy Policy für den Anwendungsfall Ambient Assisted Living.

Kapitel 11.

Schluss

In diesem Kapitel werden zunächst die Ergebnisse dieser Arbeit zusammengefasst, im Kontext der in Kapitel 3 definierten Anforderungen reflektiert und die Grenzen der vorgestellten Ansätze aufgezeigt. Ein Ausblick auf weiterführende Themen im Kontext der untersuchten Forschungsfrage und eine Zusammenfassung schließen die Arbeit ab.

11.1. Ergebnisse

Mit MontiSecArc wird in Kapitel 4 eine Security-Architekturbeschreibungssprache vorgestellt, die durch ihre Sprachelemente Grundeinheiten für die Security-Architekturmodellierung bereitstellt. Diese können Entwickler bereits im Entwurf verwenden, um wesentliche Security-Eigenschaften eines Systems festzulegen. Dazu abstrahiert MontiSecArc von Details wie dem verwendeten Verschlüsselungsverfahren mit dem die Confidentiality von Daten bei der Übertragung zwischen zwei Komponenten geschützt wird. Dadurch wird ein Gesamtüberblick über die Schutzmaßnahmen ermöglicht, die in einem System eingesetzt werden, um es gegen Angreifer zu schützen. Außerdem werden Security-Eigenschaften stets über konstruktive Sprachelemente ausgedrückt, um Anforderung A3 zu erfüllen und Entwicklern, die keine Security-Experten sind zu ermöglichen die Security-Architektur eines Systems während der Entwicklung ohne die Hilfe eines Security-Experten zu beschreiben.

Wie die in Kapitel 10 beschriebene Evaluation zeigt, modellieren Entwickler durch den Einsatz von MontiSecArc signifikant mehr Trust Level und somit potentielle Angreifer. Daher fördert MontiSecArc die Compartmentalization der Architektur und ermöglicht es Entwicklern die Vertrauensbeziehungen zwischen Komponenten während des Entwurfs zu modellieren. Ein Vergleich mit der STRIDE Analyse zeigte, dass STRIDE in MontiSecArc-Modellen mehr potentielle Fehler identifiziert, was jedoch durch die stärkere Compartmentalization begünstigt wird, da STRIDE Fehler hauptsächlich auf der Basis von Trust Levels identifiziert.

Durch den konstruktiven Charakter von MontiSecArc lassen sich die im Modell spezifizierten Schutzmaßnahmen mit Hilfe von in MontiCore realisierten Generatoren in Implementierungen übersetzen, die die spezifizierten Schutzmaßnahmen überall dort im generieren Code verwenden, wo sie in der Security-Architektur vorgesehen wurden. Diese Kopplung von Architektur und Implementierung erlaubt es die Suche nach Flaws im MontiSecArc-Modell durchzuführen und mit den Ergebnissen dieser Analyse auch eine Aussage über das implementierte System treffen zu können. Automatisierte Analysen erlauben es in einem agilen modellgetriebenen Entwicklungsprozess (Anforderung A1) Flaws bereits während der Erstellung der Security-

Architektur zu identifizieren und somit diese frühzeitig nach ihrem Entstehen zu beseitigen. Die Kombination aus automatisierter Analyse von Flaws und Generierung von Schutzmaßnahmen soll helfen, dass Anforderung A4 erfüllt wird und keine oder zumindest weniger Systeme entstehen, die bekannte Vulnerabilities enthalten. Dies wird erreicht, indem Flaws im MontiSecArc-Modell identifiziert werden und Bugs, die bei manueller Implementierung sporadisch auftreten können, durch die Generierung systematisch auftreten bzw. nicht auftreten. Wird ein Bug im generierten Code eines Generators behoben, kann dieser in allen Implementierungen behoben werden indem der aktualisierte Generator verwendet wird.

Zur Beschreibung von Flaw Correction Pattern wird aus MontiSecArc eine domänenspezifische Transformationssprache generiert, die es erlaubt Muster in MontiSecArc Syntax anzugeben, die als Flaw bekannt sind und durch eine Transformation zu beschreiben, wie ein solcher Flaw behoben wird. Diese Notation erlaubt es Flaws präzise zu modellieren und automatisiert zu prüfen, ob ein MontiSecArc-Modell einen Flaw enthält. Im Idealfall kann ein Flaw durch Anwendung der Transformation sogar automatisiert behoben werden. Durch die automatische Ausführung der Transformationen lassen sich diese ebenfalls in einem agilen modellgetriebenen Entwicklungsprozess einsetzen, um Flaws zu identifizieren und zu beheben. Mit diesem Ansatz wurden mehrere bekannte Arten von Flaws formalisiert und somit für die automatische Analyse nutzbar gemacht. Da für eine solche automatische Analyse kein Security-Experte benötigt wird, ermöglicht MontiSecArc Security auch in Softwareentwicklungsprojekten einzubauen, in denen kein Security Experte teil des Projektteams ist (Anforderung A2).

Außerdem wurde ein Generator entwickelt, der die systemspezifische Konfiguration eines Snort Intrusion Detection System aus einem MontiSecArc-Modell ableitet und somit die Überwachung eines Systems auf mögliche Netzwerkangriffe ermöglicht.

In dem in Kapitel 7 vorgestellten modellgetriebenen Security Testing Ansatz spezifizieren Entwickler über funktionale Tests das Sollverhalten einer Webapplikation. Dabei werden die Zustände einer Nutzer-Session und die für die Funktionalität der Anwendung notwendigen Übergänge zwischen diesen in einem Statechart modelliert. Dazu wurde in die UML/P Statechartsprache Java und JWebUnit Code eingebettet, sodass Entwickler Testfälle in den Aktionen von Zuständen und Transitionen spezifizieren können, die überprüfen, ob die Session tatsächlich in einem gewissen Zustand ist bzw. Transitionen zwischen Zuständen in der Webapplikation durchführen. Somit bestimmt dieses Statechart in welchen möglichen Reihenfolgen die spezifizierten JWebUnit Tests ausgeführt werden können. Dieses funktionale Modell wird dazu genutzt Testfälle, die von Entwicklern für einen spezifischen Ablauf definiert wurden in allen möglichen Abläufen zu testen. In dieser Arbeit werden die Transitionen des Statecharts nicht in der vom Entwickler vorgesehenen Reihenfolge aufgerufen, sondern getestet, ob auch Transitionen möglich sind, die bekannte Weaknesses im Session Management darstellen. Diese Weaknesses kann ein Angreifer beispielsweise dazu nutzen Aktionen in einer Applikation im Namen von anderen Benutzern auszuführen.

Da Entwickler nur funktionale Testfälle spezifizieren, welche zum Testen von Security Vulnerabilities in der Applikation verwendet werden ist Anforderung A3 erfüllt. Durch die automatisierte Verarbeitung des Statecharts mit Hilfe von MontiCore können auch die abgeleiteten Security-Tests automatisiert durchgeführt werden, sodass der Ansatz ohne zusätzlichen Aufwand für Entwickeler (A2) in einem agilen modellgetriebenen Entwicklungsprozess (A1) einsetzbar ist.

Der in Kapitel 8 vorgestellte MontiCoder Ansatz ermöglicht es bei der Spezifikation eines Kommunikationsprotokolls explizit das Encoding von Daten in diese Kommunikationssprache zu definieren. Dazu wird die kontextfreie Sprache über eine Grammatik definiert, in der eine Encoding Table für jede Produktion definiert werden kann. Diese Encoding Table gibt an, wie Daten in dieser Produktion beim Unparsen encodet und beim Parsen decodet werden müssen. Durch die explizite Definition des Encodings lässt sich mit MontiCoder ein Unparser generieren, der automatisch das für die Produktion geltende Encoding anwendet und ein Parser, der das entsprechende Decoding vornimmt.

Durch dieses automatisierte Encoding werden Injection Vulnerabilities in beliebige kontextfreie Sprachen verhindert, sobald die Sprache und das Encoding definiert wurden. Somit erfüllt MontiCoder die Anforderung A4, da durch die Verwendung des generierten Parser und Unparser Injection Vulnerabilities verhindert werden. Gleichzeitig spezifizieren Sprachentwickler in der Encoding Table konstruktiv die Schutzmaßnahmen gegen Injections, sodass auch hier Anforderung A3 erfüllt wird. Durch die automatische Generierung der Verarbeitungswerkzeuge lassen sich neue Kommunikationssprachen in einem agilen modellgetriebenen Entwicklungsprozess erstellen (Anforderung A1) und der Aufwand für die Nutzung der generierten (Un)Parser ist gering und daher in jedem Projekt möglich (Anforderung A2).

Neben Security erwarten Nutzer eines Systems auch, dass ihre Privacy gewahrt wird. Damit Entwickler diesen Aspekt bereits während des Entwurfs berücksichtigen können wird in Kapitel 9 die Privacy Development Language (PDL) vorgestellt, welche es erlaubt die Datenstruktur eines Services in einem Klassendiagramm zusammen mit einer Beschreibung der Nutzung der Daten anzugeben. Diese detaillierte Beschreibung erlaubt es eine Privacy Policy zu generieren, welche dem Nutzer bis auf Methoden und Attribute genau darstellt, zu welchem Zweck seine Daten verarbeitet werden. Solch detaillierte Informationen über die Datenverarbeitung enthalten herkömmliche Privacy Policies nicht, sodass dieser modellgetriebene Privacy Ansatz die Transparenz über die Datenverarbeitung für den Nutzer verbessert und somit Anforderung A6 erfüllt. Außerdem erlaubt es die PDL Parameter von Methoden als optional zu markieren, wodurch in der generierten Privacy Policy eine Wahlmöglichkeit für den Nutzer entsteht und die Beschreibung der Alternativen aus der PDL dort dargestellt werden. Somit wird die Entscheidung, ob eine Methode Daten erhält oder nicht, vom Entwickler an den Nutzer übertragen, was eine selbstbestimmte Nutzung des Services ermöglicht. Für Entwickler bedeutet diese Stärkung der Nutzer-Privacy eine zusätzliche Herausforderung, wobei die PDL eine konkrete Unterstützung liefert, wie Anforderung A5 dies fordert. Für die Nutzung der PDL und somit für eine Verbesserung der Nutzer-Privacy ist kein Expertenwissen notwendig (Anforderung A3) und durch die MontiCore Werkzeuginfrastruktur ist es möglich die PDL sowohl für die Realisierung der Privacy als auch zur Generierung der Implementierung in einem agilen modellgetriebenen Entwicklungsprozess zu nutzen (Anforderung A1). Durch die Verbindung von Datenmodell und Privacy-Implikationen wird des weiteren ein präzises Datenschutzaudit der Implementierung durch einen externen Auditor möglich, der durch das PDL-Modell mit geringem Aufwand die Verbindung zwischen Privacy Policy und der Implementierung herstellen kann.

11.2. Grenzen der Ansätze

Die in dieser Arbeit vorgestellten Ansätze stellen kein Patentlösung für alle Security- und Privacy-Probleme dar. Daher werden in diesem Abschnitt die bekannten Grenzen der vorgestellten Ansätze beschrieben.

Alle vorgestellten Ansätze zielen auf eine Integration in den Entwicklungsprozess ab, um bereits während der Erstellung von Software eingesetzt zu werden. Somit sind die Ansätze nicht direkt auf bestehende Systeme anwendbar, sondern aus diesen Systemen müssen zunächst die im Entwicklungsprozess verwendeten Modelle extrahiert werden und im Anschluss die handgeschriebene Implementierung Schritt für Schritt durch generierten Code abgelöst werden.

Um die Nutzung von Sprachen für Entwickler zu vereinfachen ist die Integration in eine Entwicklungsumgebung hilfreich, bei der Editoren Syntax Highlighting und Autocompletion für die verschiedenen Sprachen anbieten. Im Fokus der in dieser Arbeit vorgestellten Werkzeuge steht jedoch die automatische Verarbeitung von Modellen und die Integration in den Maven Build-Prozess.

Die vorgestellten Ansätze sind nicht zur Bevormundung von Entwicklern gedacht, sondern Entwickler müssen diese freiwillig einsetzen, nicht aktiv versuchen diese zu umgehen und bereit sein Vulnerabilities zu beheben. Die Ansätze unterstützen Entwickler dabei Fehler zu vermeiden und zu erkennen. Sie sind jedoch nicht dafür konzipiert einen Entwickler, der absichtlich Hintertüren in ein System einbaut davon abzuhalten.

Für den MontiCoder Ansatz ergibt sich durch die Unentscheidbarkeit von Aussagen über touringvollständige Sprachen, dass die Anwendung von MontiCoder in einem Prozess P_1 lediglich sicherstellen kann, dass Informationen die P_1 an einen Prozess P_2 sendet von P_2 so geparsed werden, wie P_1 diese unparsed hat. Wenn die Informationen beim Empfänger P_2 jedoch interpretiert werden und somit eine ausführbare Semantik besitzen, die eine Berechnung und Ausgabe von Informationen zulässt, kann nicht allein durch den Einsatz von MontiCoder in P_1 sichergestellt werden, dass keine Injection Vulnerability entsteht. Daher muss der Empfänger bei der Ausführung der erhaltenen Informationen sicherstellen, dass erneut ein Unparser verwendet wird, der ein korrektes Encoding durchführt. Dies begründet warum jeder Parser und Unparser in jeder Komponente eines Systems, die direkt oder indirekt Eingabedaten verarbeitet, relevant für die Security des Gesamtsystems ist.

Modellgetriebene Privacy erhöht den Detaillierungsgrad der Informationen, die Servicenutzer über die Verarbeitung ihrer Daten erfahren. Technisch sind Serviceanbieter demnach in der Lage die Datennutzung genauestens offenzulegen. Es stellt sich jedoch die Frage in wieweit dies in ihrem Interesse liegt, wenn sie durch die Nutzung von Daten abseits des Services erhebliche Gewinne erzielen und diese Datennutzung durch modellgetriebene Privacy offenbart wird. An diese Frage der Akzeptanz bei Servicebetreibern schließt sich an, ob Nutzer aus der detaillierten Auflistung von Konsequenzen der Nutzung und nicht-Nutzung von Teilservices den Wert ihrer Daten ableiten und unter Umständen ihr Verhalten aufgrund dieser Erkenntnis verändern.

11.3. Weiterführende Arbeiten im Kontext der Forschungsfrage

Aus den hier vorgestellten Ansätzen ergeben sich weitere interessante Fragestellungen für die Forschung und auch im Bezug auf die Anwendbarkeit und den Nutzen der Vorgestellten Verfahren.

Ziel von MontiSecArc ist es, dass Entwickler alle für die Security-Architektur relevanten Eigenschaften modellieren können. Unter welchen Bedingungen diese Eigenschaften kompositional sind würde die Beschreibung der Eigenschaften in der Security-Architektur unter Umständen vereinfachen, da durch die Kompositionalität weniger Eigenschaften explizit angegeben werden müssen und mehr abgeleitet werden können.

Die Beschreibung von Flaw Correction Pattern als Transformation auf einem MontiSecArc-Modell erlaubt die präzise Definition von Flaws und somit eine Diskussion über bekannte Flaw Correction Pattern und eine einheitliche Benennung der Flaws. Eine allgemein anerkannte Sammlung solcher Flaw Correction Pattern in einem Katalog könnte genutzt werden, um eine Security-Architektur gegen diese Sammlung zu testen und somit einen Standard für die Prüfung von Security-Architekturen zu setzen. Diese Prüfung stellt somit ein Gütesiegel dar, welches automatisiert überprüfbar und daher objektiv ist.

Einige Flaw Correction Pattern beschreiben Zusammenhänge zwischen Komponenten, die nicht direkt verbunden sind, sondern durch einen Pfad. Die hier verwendete generierte Transformationssprache besitzt noch kein Sprachelement, welches es erlaubt diese Zusammenhänge zu formulieren. Daher erscheint die Erweiterung der generierten Sprache oder die Erweiterung der Generierung der Transformationssprachen um ein solches Sprachelement als sinnvoll.

Durch die Systematisierung von Flaws als Flaw Correction Pattern ergibt sich eine neue Abstraktionsstufe auf der Security in der Architektur ausgedrückt werden kann. Wenn es gelingt ein Maß zu definieren, welches die positiven und negativen Eigenschaften auf dieser Abstraktionsebene in Zahlen umwandelt, lassen sich Security-Architekturen anhand dieser Metrik vergleichen.

Um Injection Vulnerabilities zu verhindern müssen Entwickler alle Sprachen definieren, die zur Kommunikation in einem System verwendet werden. Solange das Erstellen einer Grammatik der Ausgabesprache eines Programms wesentlich komplexer ist als Strings zu konkatenieren und auszugeben werden Entwickler weiterhin lieber ohne formale Definition der Ausgabesprache arbeiten. Daher sind Ansätze notwendig, mit denen Entwickler die Struktur von Ein- und Ausgabedaten eines Programms einfacher beschreiben können, um auf diese Definition der Sprachen den LangSec Ansatz anzuwenden und Injection Vulnerabilities in allen Programmen zu verhindern.

11.4. Zusammenfassung

In dieser Arbeit wurden mehrere Ansätze vorgestellt, die zeigen, dass Modelle sowohl zur Abstraktion als auch zur Präzisierung eingesetzt werden können, um Software Security und Privacy in Systeme einzubauen, nachzuweisen und zu testen. Durch einen agilen modellgetriebenen Entwicklungsprozess in dem Modelle und Implementierung verbunden werden, können Konzepte, die Security und Privacy sicherstellen auf Modellebene eingebaut und überprüft werden, anstatt diese bei einer Überprüfung nach der Entwicklung aus der Implementierung zu extrahieren.

Mit MontiSecArc wurde eine Security-Architekturbeschreibungssprache vorgestellt, die von Entwicklern bereits im Entwurf zur Definition der Security-Architektur eines Systems verwendet wird. MontiSecArc-Modelle lassen sich automatisiert auf bekannte Flaw Correction Pattern testen, sodass Entwickler direkt auf bekannte Flaws hingewiesen werden, bevor sich diese in der Implementierung fortpflanzen. Zur Beschreibung von Flaw Correction Pattern wurde eine aus MontiSecArc generierte domänenspezifische Transformationssprache verwendet, die es Security-Experten erlaubt Flaws zu spezifizieren und in der Literatur bekannte allgemeine Prinzipien zu konkretisieren. Außerdem wurden Generatoren vorgestellt, die in MontiSecArc spezifizierte Schutzmaßnahmen in eine Java Implementierung übersetzen und zusätzlich ein Network Intrusion Detection System generieren.

Zum Testen von bekannten Vulnerabilities im Session Management von Webapplikationen wurde ein modellgetriebener Security Testing Ansatz vorgestellt, bei dem Entwickler die Funktionalität einer Webapplikation über JWebUnit Testfälle testen und diese in ein Statechart einbetten. Dieses Statechart bestimmt in welcher Reihenfolge Aktionen der Webseite vom Nutzer ausgeführt werden dürfen.

Zur Verhinderung von weit verbreiteten Injection Angriffen in beliebigen kontextfreien Sprachen wurde der MontiCoder Ansatz vorgestellt, bei dem das Encoding und Decoding einer Sprache, die zur Kommunikation zwischen zwei Prozessen verwendet wird, explizit in die Grammatik integriert wird. Aus dieser Sprachdefinition generiert MontiCoder Parser und Unparser, die einen korrekten Round-Trip für beliebig maliziöse Daten in einem AST sicherstellen, wenn die Definition des Encodings in der Sprachdefinition korrekt ist. Durch die automatische Anwendung des kontextsensitiven Encodings im generierten Unparser wird die Entstehung von Injection Vulnerabilities verhindert, da Entwickler nicht das korrekte kontextspezifische Encoding für jede Ausgabe manuell bestimmen müssen.

Mit modellgetriebener Privacy wurde ein Ansatz vorgestellt, der es Entwicklern erlaubt die Privacy von Nutzern bereits während des Entwurf des Datenmodell eines Services zu berücksichtigen. Dabei verwendeten Entwickler die Privacy Development Language (PDL), um den Zweck der Verarbeitung von Nutzerdaten zu beschreiben. Es wurde gezeigt, wie aus einem PDL-Modell eine interaktive Privacy Policy generiert wird, die es einem Nutzer erlaubt zu entscheiden, welche Daten er an einen Service übertragen möchte. Die aus dieser Entscheidung resultierenden Konsequenzen werden ihm ebenfalls in der Privacy Policy anzeigt. Dieser Ansatz verlagert die Entscheidung über die Verwendung von Nutzerdaten vom Entwickler zum Nutzer, wodurch der Nutzer die Möglichkeit erhält selbstbestimmte, informierte Entscheidungen über seine individuelle Privacy zu treffen. Durch die Verbindung von Aussagen aus der Privacy Policy mit Methoden und Attributen des Datenmodells eines Services, welche die PDL herstellt wird zudem die Auditierung der Implementierung vereinfacht, bei der ein unabhängiger Dritter überprüft, ob die Implementierung das in der Privacy Policy angegebene Verhalten auch tatsächlich besitzt.

Die vorgestellten Ansätze wurden in Fallstudien anhand von Beispielen aus den Forschungsprojekten SensorCloud und IPACS demonstriert. Der Einsatz von MontiSecArc bei der Entwicklung einer Security-Architektur wurde in einer Evaluierung gegen eine Kontrollgruppe verglichen, die mit herkömmlichen Architekturbeschreibungssprachen gearbeitet hat. Dabei konnte gezeigt werden, dass durch den Einsatz von MontiSecArc Entwickler mehr Trust Level modellierten und somit Security-Architekturen entstehen, die eine stärkere Compartmentalization besitzen.

Literaturverzeichnis

[AAB10] Marwan Abi-Antoun and Jeffrey M. Barnes. Analyzing Security Architectures. In *International Conference on Automated Software Engineering (ASE)*, pages 3–12. ACM, 2010.

[Acu15] Acunetix. Acunetix. `https://www.acunetix.com/`, 2015. [Online; Zugriff 13.11.2015].

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[AFHOT06] Jim Alves-Foss, W. Scott Harrison, Paul W. Oman, and Carol Taylor. The MILS Architecture for High-Assurance Embedded Systems. *International Journal of Embedded Systems*, 2(3-4):239–247, 2006.

[AGI13] Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. Automated Software Architecture Security Risk Analysis Using Formalized Signatures. In *International Conference on Software Engineering (ICSE)*, pages 662–671, May 2013.

[Aho03] Alfred V Aho. *Compilers: Principles, Techniques and Tools*. Pearson Education India, 2003.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabit. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid F. King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, 1977.

[AK97] Ross Anderson and Markus Kuhn. Low Cost Attacks on Tamper Resistant Devices. In Bruce Christianson, Bruno Crispo, Mark Lomas, and Michael Roe, editors, *International Workshop on Security Protocols*, volume 1361 of *LNCS*, pages 125–136, Paris, April 1997. Springer.

[Ale03] Ian Alexander. Misuse Cases: Use Cases with Hostile Intent. *IEEE Software*, 20(1):58–66, January 2003.

[Apa15a] Apache Software Foundation. Maven. `http://maven.apache.org/`, 2015. [Online; Zugriff 13.11.2015].

[Apa15b] Apache Software Foundation. Shiro. `http://shiro.apache.org/`, 2015. [Online; Zugriff 13.11.2015].

[AvdBS11] B. J. Arnoldus, Mark G. J. van den Brand, and Alexander Serebrenik. Less is More: Unparser-completeness of Metalanguages for Template Engines. In *International Conference on Generative Programming and Component Engineering (GPCE)*, pages 137–146. ACM, 2011.

[Bar11] A. Barth. RFC6265: HTTP State Management Mechanism, 2011.

[BB01] Barry W. Boehm and Victor R. Basili. Software Defect Reduction Top 10 List. *IEEE Computer*, 34(1):135–137, 2001.

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Agile Manifesto. `http://agilemanifesto.org/`, 2001. [Online; Zugriff 13.11.2015].

[BdDPP14] Alessio Botta, Walter de Donato, Valerio Persico, and Antonio Pescapé. On the Integration of Cloud Computing and Internet of Things. In *International Conference on Future Internet of Things and Cloud*, pages 23–28. IEEE, 2014.

[BDL+14] Sergey Bratus, Trey Darley, Michael Locasto, Meredith L. Patterson, Rebecca bx Shapiro, and Anna Shubina. Beyond Planted Bugs in "Trusting Trust": The Input-Processing Frontier. *IEEE Security & Privacy*, 12(1):83–87, January 2014.

[BDM+04] Jay Beale, Renaud Deraison, Haroon Meer, Roelof Temmingh, and Charl Van Der Walt. *Nessus Network Auditing*. Syngress Publishing, 2004.

[BDV07] Martin Bravenboer, Eelco Dolstra, and Eelco Visser. Preventing injection attacks with syntax embeddings. In *International Conference on Generative Programming and Component Engineering (GPCE)*, pages 3–12. ACM, 2007.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[Bec03] Richard Beckwith. Designing for Ubiquity: The Perception of Privacy. *IEEE Pervasive Computing*, 2(2):40–46, 2003.

[BEJV96] Pam Binns, Matt Engelhart, Mike Jackson, and Seve Vestal. Domain-Specific Software Architectures for Guidance, Navigation and Control. *International Journal of Software Engineering and Knowledge Engineering*, 06(02):201–227, 1996.

[BH13] Anya H. Bagge and Tero Hasu. A Pretty Good Formatting Pipeline. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering (SLE)*, volume 8225 of *LNCS*, pages 177–196. Springer, 2013.

[BJN07] Bastian Best, Jan Jürjens, and Bashar Nuseibeh. Model-Based Security Engineering of Distributed Information Systems Using UMLsec. In *International Conference on Software Engineering (ICSE)*, pages 581–590. IEEE, 2007.

[BKK06] Carolyn A. Brodie, Clare-Marie Karat, and John Karat. An Empirical Study of Natural Language Parsing of Privacy Policy Rules Using the SPARCLE Policy Workbench. In *Symposium on Usable Privacy and Security*, pages 8–19. ACM, 2006.

[BKK11] Marianne Busch, Alexander Knapp, and Nora Koch. Modeling Secure Navigation in Web Information Systems. In Janis Grabis and Marite Kirikova, editors, *International Conference on Business Perspectives in Informatics Research*, LNBIP, pages 239–253. Springer, 2011.

[BKKF05] Carolyn A. Brodie, Clare-Marie Karat, John Karat, and Jinjuan Feng. Usable Security and Privacy: A Case Study of Developing Privacy Management Tools. In *Symposium on Usable Privacy and Security*, pages 35–43. ACM, 2005.

[BL73] David E. Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-73-278, Vol. 1, MITRE, 1973.

[Blo08] Joshua Bloch. *Effective Java*. Addison-Wesley, 2nd edition, 2008.

[Boe88] Barry W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[BOS13] Marianne Busch, Martín Ochoa, and Roman Schwienbacher. Modeling, Enforcing and Testing Secure Navigation Paths for Web Applications. Technical Report 1301, Ludwig-Maximilians-Universität München, 2013.

[BPH13] Sergey Bratus, Meredith L. Patterson, and Dan Hirsch. From “shotgun parsers” to more secure stacks. In *Shmoocon*, November 2013.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer, 2001.

[BSJ11] Koen Buyens, Riccardo Scandariato, and Wouter Joosen. Least privilege analysis in software architectures. *Software & Systems Modeling (SoSyM)*, 12(2):331–348, 2011.

[BSK16] Bernhard J. Berger, Karsten Sohr, and Rainer Koschke. Automatically extracting threats from extended data flow diagrams. In Juan Caballero, Eric Bodden, and Elias Athanasopoulos, editors, *8th International Symposium on Engineering Secure Software and Systems (ESSoS)*, pages 56–71, Cham, April 2016. Springer.

[Bur05] Jesse Burns. Cross Site Reference Forgery: An introduction to a common web application weakness. 2005.

[BV08] Prithvi Bisht and V.N. Venkatakrishnan. XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, volume 5137 of *LNCS*, pages 23–43. Springer, 2008.

[BW07] Thomas Baar and Jon Whittle. On the Usage of Concrete Syntax in Model Transformation Rules. In *International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI)*, pages 84–97. Springer, 2007.

[Cav16] Ann Cavoukian. Privacy by Design The 7 Foundational Principles. `http://www.ipc.on.ca/images/Resources/7foundationalprinciples.pdf`, 2016. [Online; Zugriff 13.02.2016].

[CF12] Pietro Colombo and Elena Ferrari. Towards a Modeling and Analysis Framework for Privacy-Aware Systems. In *International Confernece on Social Computing (SocialCom) and International Conference on Privacy, Security, Risk and Trust (PASSAT)*, pages 81–90. IEEE, 2012.

[Cis15] Cisco. Snort. `https://www.snort.org`, 2015. [Online; Zugriff 13.11.2015].

[Cla16] James R. Clapper. Worldwide Thread Assessment of the US Intelligence Community. `http://www.intelligence.senate.gov/sites/default/files/wwt2016.pdf`, 2016. [Online; Zugriff 13.02.2016].

[CM04] Brian Chess and Gary McGraw. Static Analysis for Security. *IEEE Security & Privacy*, 2(6):76–79, November 2004.

[Coh87] Fred Cohen. Computer Viruses. *Computers & Security*, 6(1):22 – 35, 1987.

[Com91] Commission of the European Communities. *Information Technology Security Evaluation Criteria (ITSEC)*. Department of Trade and Industry, 1991.

[Cou13] Martin Courtney. Premium binds. *IEEE Engineering & Technology*, 8(6):68–73, 2013.

[CvdM15] Stephen Chong and Ron van der Meyden. Using Architecture to Reason About Information Security. *ACM Transactions on Information and System Security (TISSEC)*, 18(2):8:1–8:30, December 2015.

[Dan13] Nils A. Danielsson. Correct-by-construction Pretty-printing. In *ACM SIGPLAN Workshop on Dependently-typed Programming (DTP)*, pages 1–12. ACM, 2013.

[De 13] De Brauw Blackstone Westbroek N.V. EU Country Guide Data Location & Access Restriction, 2013.

[Del15a] Jim DelGrosso. Architecture Analysis. `https://handouts.secappdev.org/handouts/2015/Jim%20DelGrosso/2015-02-23,%20SecAppDev,%20Architecture%20Analysis.pdf`, 2015. [Online; Zugriff 13.11.2015].

[Del15b] Jim DelGrosso. Threat Modeling. `https://handouts.secappdev.org/handouts/2015/Jim%20DelGrosso/2015-02-24,%20SecAppDev,%20Threat%20Modeling.pdf`, 2015. [Online; Zugriff 13.11.2015].

[Die07] Andreas Diekmann. *Empirische Sozialforschung*. Rowohlt Verlag, Hamburg, 2te edition, 2007.

[DP09] Stephane Ducasse and Damien Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.

[DR08] T. Dierks and E. Rescorla. RFC5246: The Transport Layer Security (TLS) Protocol Version 1.2, 2008.

[DvdHT02] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *International Conference on Software Engineering (ICSE)*, pages 266–276, May 2002.

[DWTB03] Yi Deng, Jiacun Wang, Jeffrey J.P. Tsai, and Konstantin Beznosov. An Approach for Modeling and Analysis of Security System Architectures. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1099–1119, September 2003.

[EB10] Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA)*, pages 307–309. ACM, 2010.

[Eck14] Claudia Eckert. *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. De Gruyter Studium. Oldenbourg Wissenschaftsverlag, 9te edition, 2014.

[EHH+14] Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle. SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators. In Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe, editors, *Trusted Cloud Computing*, pages 203–218. Springer, 2014.

[EHKR14] Michael Eggert, Roger Häußling, Daniel Kerpen, and Kirsten Rüssmann. SensorCloud: Sociological Contextualization of an Innovative Cloud Platform. In Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe, editors, *Trusted Cloud Computing*, pages 295–313. Springer, 2014.

[EvdSV+13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, WilliamR. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin

van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The State of the Art in Language Workbenches. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, volume 8225 of *LNCS*, pages 197–217. Springer, 2013.

[FGH06] Peter Feiler, David Gluch, and John Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 2006.

[FKC07] David Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House, 2nd edition, 2007.

[FKK+14] Michael Felderer, Basel Katt, Philipp Kalb, Jan Jürjens, Martín Ochoa, Federica Paci, Le Minh Sang Tran, Thein Than Tun, Koen Yskout, Riccardo Scandariato, Frank Piessens, Dries Vanoverberghe, Elizabeta Fourneret, Matthias Gander, Bjørnar Solhaug, and Ruth Breu. Evolution of Security Engineering Artifacts: A State of the Art Survey. *International Journal of Secure Software Engineering (IJSSE)*, 5(4):48–98, 2014.

[FLN+12] Shamal Faily, John Lyle, Cornelius Namiluko, Andrea Atzeni, and Cesare Cameroni. Model-driven Architectural Risk Analysis Using Architectural and Contextualised Attack Patterns. In *Workshop on Model-Driven Security (MDsec)*, pages 3:1–3:6. ACM, 2012.

[FMP05] Eduardo Fernández-Medina and Mario Piattini. Designing Secure Databases. *Information and Software Technology*, 47(7):463 – 477, 2005.

[FMTVP07] Eduardo Fernández-Medina, Juan Trujillo, Rodolfo Villarroel, and Mario Piattini. Developing Secure Data Warehouses With a UML Extension. *Information Systems*, 32(6):826 – 856, 2007.

[FRe14a] R. Fielding and J. Reschke editors. RFC7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, 2014.

[FRe14b] R. Fielding and J. Reschke editors. RFC7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, 2014.

[Fre15] Freemarker project. Freemarker. `http://freemarker.org/`, 2015. [Online; Zugriff 13.11.2015].

[Gal04] Daniel Galin. *Software Quality Assurance: From Theory to Implementation*. Pearson education, 2004.

[Gar15] Gartner Inc. Gartner Says Worldwide Security Software Market Grew 5.3 Percent in 2014. `http://www.gartner.com/newsroom/id/3062017`, 2015. [Online; Zugriff 13.11.2015].

[Gee03] Dan Geer. Risk Management Is Still Where the Money Is. *IEEE Computer*, 36(12):129–131, Dec 2003.

[GF14] Martin Glinz and Samuel A. Fricker. On Shared Understanding in Software Engineering: An Essay. *Computer Science - Research and Development*, 30(3):363–376, 2014.

[GG12] Jeffrey Gennari and David Garlan. Measuring Attack Surface in Software Architecture. Technical Report CMU-ISR-11-121, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2012.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[GHK+15] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. A Comparison of Mechanisms for Integrating Handwritten and Generated Code for Object-Oriented Programming Languages. In *Model-Driven Engineering and Software Development Conference (MODELSWARD'15)*, pages 74–85, 2015.

[Gid13] Mirren Gidda. Edward Snowden and the NSA files – timeline. `http://www.theguardian.com/world/2013/jun/23/edward-snowden-nsa-files-timeline`, August 2013. [Online; Zugriff 13.11.2015].

[Gre15] Steffen Greber. Implementierung eines Intrusion Detection System Generators für MontiSecArc. Bachelorarbeit, Rheinisch Westfälische Technische Hochschule Aachen, 2015.

[Grø09] Roy Grønmo. *Using Concrete Syntax in Graph-based Model Transformations*. PhD thesis, University of Oslo, 2009.

[GTD11] Seda Gürses, Carmela Troncoso, and Claudia Diaz. Engineering Privacy by Design. In *Conference on Computers, Privacy & Data Protection*, pages 25–28, 2011.

[Hab16] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Number 24 in Aachener Informatik-Berichte, Software Engineering. Shaker Verlag, 2016.

[Har88] Norm Hardy. The Confused Deputy: (or Why Capabilities Might Have Been Invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, October 1988.

[He03] Qingfeng He. Privacy Enforcement with an Extended Role-Based Access Control Model. Technical Report TR-2003-09, Department of Computer Science, North Carolina State University, 2003.

[Hew15] Hewlett Packard. HP Fortify. http://www8.hp.com/us/en/software-solutions/application-security/index.html, 2015. [Online; Zugriff 13.11.2015].

[HFM08] Jörgen Hansson, Peter H. Feiler, and John Morley. Building Secure Systems using Model-Based Engineering and Architectural Models. Technical report, Software Engineering Intitute, Carnegie Mellon University, 2008.

[HHCW12] René Hummen, Martin Henze, Daniel Catrein, and Klaus Wehrle. A Cloud Design for User-controlled Storage and Processing of Sensor Data. In *International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 232–240. IEEE, 2012.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 191–196. IEEE, 2014.

[HHK+16] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701 – 718, 2016.

[HHM+13] M. Henze, R. Hummen, R. Matzutt, D. Catrein, and K. Wehrle. Maintaining User Control While Storing and Processing Sensor Data in the Cloud. *International Journal of Grid and High Performance Computing (IJGHPC)*, 5(4):97–112, 2013.

[HHMW14] Martin Henze, Rene Hummen, Roman Matzutt, and Klaus Wehrle. A Trust Point-based Security Architecture for Sensor Data in the Cloud. In Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe, editors, *Trusted Cloud Computing*, pages 77–106. Springer, 2014.

[HHRW15] Lars Hermerschmidt, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. Generating Domain-Specific Transformation Languages for Component & Connector Architecture Descriptions. In *Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp)*, volume 1463 of *CEUR Workshop Proceedings*, pages 18–23, 2015.

[HHW13] Martin Henze, Rene Hummen, and Klaus Wehrle. The Cloud Needs Cross-Layer Data Handling Annotations. In *Security and Privacy Workshops (SPW)*, pages 18–22. IEEE, 2013.

[Hic15] Sandra Hicks. Formalization and Automated Analysis of Security Patterns in MontiSecArc Architectures. Bachelorarbeit, Rheinisch Westfälische Technische Hochschule Aachen, 2015.

[HKR15] Lars Hermerschmidt, Stephan Kugelmann, and Bernhard Rumpe. Towards More Security in Data Exchange: Defining Unparsers with Context-Sensitive Encoders for Context-Free Grammars. In *Security and Privacy Workshops (SPW)*, pages 134–141. IEEE, 2015.

[HKW+08] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. CoCoME - The Common Component Modeling Example. In Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and František Plášil, editors, *The Common Component Modeling Example*, volume 5153 of *LNCS*, pages 16–53. Springer, 2008.

[HL02] Michael Howard and David E. Leblanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2nd edition, 2002.

[HL06] Michael Howard and Steve Lipner. *The Security Development Lifecycle*. Microsoft Press, Redmond, WA, USA, 2006.

[HMA04] Paco Hope, Gary McGraw, and Annie I. Anton. Misuse and Abuse Cases: Getting Past the Positive. *IEEE Security & Privacy*, 2(3):90–92, May 2004.

[HNPR14] Lars Hermerschmidt, Antonio Navarro Pérez, and Bernhard Rumpe. A Model-based Software Development Kit for the SensorCloud Platform. In Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe, editors, *Trusted Cloud Computing*, pages 125–140. Springer, 2014.

[HP05] Robert J. Hansen and Meredith L. Patterson. Guns and Butter: Towards Formal Axioms of Input Validation. In *Black Hat USA*, 2005.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRW15a] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 136–145. IEEE, 2015.

[HRW15b] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[HSS14] Bernhard Hoisl, Stefan Sobernig, and Mark Strembeck. Modeling and enforcing secure object flows in process-driven SOAs: an integrated model-driven approach. *Software & Systems Modeling (SoSyM)*, 13(2):513–548, 2014.

[Hug95] John Hughes. The design of a pretty-printing library. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*, pages 53–96. Springer, 1995.

[Hug15] Jérôm Hugues. Ocarina AADL model processor. `http://www.openaadl.org/ocarina.html`, 2015. [Online; Zugriff 13.11.2015].

[HWF+10] Jörgen. Hansson, Lutz Wrage, Peter H. Feiler, John Morley, Bruce Lewis, and Jérôm Hugues. Architectural Modeling to Verify Security and Nonfunctional Behavior. *IEEE Security & Privacy*, 8(1):43–49, January 2010.

[IAD+14] IEEE Computer Society Center for Secure Design, Iván Arce, Neil Daswani, Jim DelGrosso, Danny Dhillon, Christoph Kern, Tadayoshi Kohno, Carl Landwehr, Gary McGraw, Brook Schoenfield, Margo Seltzer, Diomidis Spinellis, Izar Tarandach, and Jacob West. Avoiding the Top 10 Software Security Design Flaws. 2014.

[IM09] Kyungsoo Im and John D. McGregor. Debugging Support for Security Properties of Software Architectures. In *Cyber Security and Information Intelligence Research Workshop (CSIIRW)*, pages 16:1–16:4. ACM, 2009.

[Int15] Internet Security Research Group. Let's Encrypt. `https://letsencrypt.org/`, 2015. [Online; Zugriff 13.11.2015].

[ISKC11] Iulia. Ion, Niharika. Sachdeva, Ponnurangam Kumaraguru, and Srdjan Capkun. Home is Safer than the Cloud! Privacy Concerns for Consumer Cloud Storage. In *Symposium on Usable Privacy and Security*, pages 13:1–13:20. ACM, 2011.

[ISO11a] ISO. Information technology — Security techniques — Application security. ISO 27034, International Organization for Standardization, Geneva, Switzerland, 2011.

[ISO11b] ISO. Systems and software engineering — Architecture description. ISO 42010, International Organization for Standardization, Geneva, Switzerland, 2011.

[ISO13] ISO. Information technology — Security techniques — Information security management systems — Requirements. ISO 27001, International Organization for Standardization, Geneva, Switzerland, 2013.

[JBA13] DN Jutla, Peter Bodorik, and Sohail Ali. Engineering Privacy for Big Data Apps with the Unified Modeling Language. In *2013 IEEE International Congress on Big Data (BigData Congress)*, pages 38–45. IEEE, 2013.

[JS07] Jan Jürjens and Pasha Shabalin. Tools for secure systems development with UML. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5):527–544, 2007.

[JSB08] Jan Jürjens, Joerg Schreck, and Peter Bartmann. Model-based Security Analysis for Mobile Communications. In *International Conference on Software Engineering (ICSE)*, pages 683–692. ACM, 2008.

[Jür02] Jan Jürjens. UMLsec: Extending UML for Secure Systems Development. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *«UML» 2002 — The Unified Modeling Language*, volume 2460 of *LNCS*, pages 412–425. Springer, 2002.

[Jür05a] Jan Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[Jür05b] Jan Jürjens. Sound methods and effective tools for model-based security engineering with uml. In *International Conference on Software Engineering (ICSE)*, pages 322–331, May 2005.

[JWe15] JWebUnit. JWebUnit. `https://jwebunit.github.io/jwebunit/`, 2015. [Online; Zugriff 13.11.2015].

[Kal00] B. Kaliski. RFC2898: PKCS #5: Password-Based Cryptography Specification Version 2.0, 2000.

[KETEH06] Darrell M. Kienzle, Matthew C. Elder, David Tyree, and James Edwards-Hewitt. Security Patterns Repository, Version 1.0. 2006.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop*, volume B-108 of *Techreport*, pages 7–13. Helsinki School of Economics, October 2009.

[Kol02] Mitja Kolšek. Session fixation vulnerability in web-based applications, 2002.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Number 1 in Aachener Informatik-Berichte, Software Engineering. Shaker Verlag, März 2010.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[KS02] Günter. Karjoth and Matthias Schunter. A Privacy Policy Model for Enterprises. In *Computer Security Foundations Workshop (CSFW)*, pages 271–281. IEEE, 2002.

[Kug15] Stephan Kugelmann. Generierung von kontextabhängigen Encodern aus MontiCore Grammatiken. Masterarbeit, Rheinisch Westfälische Technische Hochschule Aachen, 2015.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman, Boston, MA, USA, 2003.

[LBD02] Torsten Lodderstedt, David Basin, and Jürgen Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *«UML» 2002 — The Unified Modeling Language*, volume 2460 of *LNCS*, pages 426–441. Springer, 2002.

[LBS09] Michael E. Locasto, Sergey Bratus, and Brian Schulte. Bickering In-Depth: Rethinking the Composition of Competing Security Systems. *IEEE Security & Privacy*, 7(6):77–81, November 2009.

[Lee08] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, 2008.

[Les12] Michael Lesk. The Price of Privacy. *IEEE Security & Privacy*, 10(5):79–81, 2012.

[LHL15] Jianghua Liu, Xinyi Huang, and Joseph K. Liu. Secure sharing of Personal Health Records in cloud computing: Ciphertext-Policy Attribute-Based Signcryption. *Future Generation Computer Systems*, 52:67 – 76, 2015.

[LMS+11] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda. *The CERT Oracle Secure Coding Standard for Java*. SEI Series in Software Engineering. Addison-Wesley Professional, 2011.

[LMS+13] Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda. *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs*. SEI Series in Software Engineering. Addison-Wesley Professional, 2013.

[LR09] Felix “FX” Lindner and Recurity Labs. Blitzableiter. `https://github.com/rtezli/Blitzableiter`, 2009.

[LSS10] Mass Soldal Lund, Bjørnar Solhaug, and Ketil Stølen. *Model-driven risk analysis: the CORAS approach*. Springer Science & Business Media, 2010.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[McG03] Gary McGraw. From the Ground Up: The DIMACS Software Security Workshop. *IEEE Security & Privacy*, 1(2):59–66, March 2003.

[McG06] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[MdAY14] Yod-Samuel Martín, Jose M. del Alamo, and Juan C. Yelmo. Engineering Privacy Requirements Valuable Lessons from Another Realm. In *Evolving Security and Privacy Requirements Engineering (ESPRE)*, pages 19–24. IEEE, August 2014.

[MG07] Haralambos Mouratidis and Paolo Giorgini. Secure Tropos: A Security-oriented Extension of the Tropos Methodology. *International Journal of Software Engineering and Knowledge Engineering*, 17(2):285–309, April 2007.

[Mic15] Microsoft. SDL Threat Modeling Tool. `https://www.microsoft.com/en-us/sdl/adopt/threatmodeling.aspx`, 2015. [Online; Zugriff 13.11.2015].

[MIT15a] MITRE. Common Attack Pattern Enumeration and Classification. `https://capec.mitre.org`, 2015. [Online; Zugriff 13.11.2015].

[MIT15b] MITRE. Common Weakness Enumeration. `https://cwe.mitre.org/`, 2015. [Online; Zugriff 13.11.2015].

[MK05] Russell A. McClure and Ingolf H. Krüger. SQL DOM: Compile Time Checking of Dynamic SQL Statements. In *International Conference on Software Engineering (ICSE)*, pages 88–96, May 2005.

[MKSK10] Philip Mayer, Nora Koch, Andreas Schroeder, and Alexander Knapp. The UML4SOA Profile. Technical report, Ludwig-Maximilians-Universität München, 2010.

[MLM+13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, June 2013.

[MMD+14] Mukhtiar Memon, Gordhan D. Menghwar, Mansoor H. Depar, Akhtar A. Jalbani, and Waqar M. Mashwani. Security modeling for service-oriented systems using security pattern refinement approach. *Software & Systems Modeling (SoSyM)*, 13(2):549–572, 2014.

[Moc87] Mockapetris, P. RFC1034: Domain Names - Concepts and Facilities, 1987.

[MQRG97] Mark Moriconi, Xiaolei Qian, Robert. A. Riemenschneider, and Li Gong. Secure Software Architectures. In *Symposium on Security and Privacy*, pages 84–93. IEEE, May 1997.

[MR97] Mark Moriconi and Robert. A. Riemenschneider. Introduction to SASDL 1.0 A Language for Specifying Software Architecture Hierarchies. Technical Report SRI-CSL-97-1, Computer Science Laboratory SRO International, March 1997.

[MT00] Nenad Medvidovic and Richad N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[MW11] Pratyusa K. Manadhata and Jeannette M. Wing. An Attack Surface Metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, May 2011.

[MW13] Kazutaka Matsuda and Meng Wang. FliPpr: A Prettier Invertible Printing System. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, volume 7792 of *LNCS*, pages 101–120. Springer, 2013.

[Nat13] National Institute of Standards and Technology. *NIST Special Publication 800-53 Revision 4: Security and Privacy Controls for Federal Information Systems and Organizations*. CreateSpace, Paramount, CA, 2013.

[Nat15] National Institute of Standards and Technology. National Vulnerability Database. `https://nvd.nist.gov/`, 2015. [Online; Zugriff 13.11.2015].

[NBL+10] Qun Ni, Elisa Bertino, Jorge Lobo, Carolyn A. Brodie, Clare-Marie Karat, John Karat, and Alberto Trombeta. Privacy-Aware Role-Based Access Control. *ACM Transactions on Information and System Security (TISSEC)*, 13(3):24:1–24:31, 2010.

[NS78] Roger M. Needham and Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, 1978.

[NTIO05] Yuichi Nakamura, Michiaki Tatsubori, Takeshi Imamura, and Koichi Ono. Model-driven security based on a Web services security architecture. In *International Conference on Services Computing (SCC)*, volume 1, pages 7–15 vol.1. IEEE, July 2005.

[OAS07] OASIS. Web Services Business Process Execution Language Version 2.0. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`, 2007. [Online; Zugriff 13.11.2015].

[OAS12] OASIS. WS-SecurityPolicy 1.3. `http://docs.oasis-open.org/ws-sx/ws-securitypolicy/v1.3/errata01/os/ws-securitypolicy-1.3-errata01-os-complete.pdf`, 2012. [Online; Zugriff 13.11.2015].

[OAS13] OASIS. eXtensible Access Control Markup Language (XACML) Version 3.0. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html`, 2013. [Online; Zugriff 13.11.2015].

[Obj14a] Object Management Group. MDA Guide revision 2.0. `http://www.omg.org/cgi-bin/doc?ormsc/14-06-01`, 2014. [Online; Zugriff 13.11.2015].

[Obj14b] Object Management Group. Object Constraint Language. `http://www.omg.org/spec/OCL/2.4/PDF`, 2014. [Online; Zugriff 13.11.2015].

[Obj15] Object Management Group. Unified Modeling Language Version 2.5. `http://www.omg.org/spec/UML/2.5/PDF`, 2015. [Online; Zugriff 13.11.2015].

[OGA05] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. MulVAL: A Logic-based Network Security Analyzer. In *USENIX Security Symposium*, pages 119 – 128, Berkeley, CA, USA, 2005. USENIX Association.

[OIML15] Open Web Application Security Project, Jeff Ichnowski, Jim Manico, and Jeremy Long. OWASP Java Encoder Project. `https://www.owasp.org/index.php/OWASP_Java_Encoder_Project`, 2015.

[Ope15a] Open Web Application Security Project. Injection Theory. `https://www.owasp.org/index.php/Injection_Theory`, 2015. [Online; Zugriff 13.11.2015].

[Ope15b] Open Web Application Security Project. OWASP Guide Project. `https://www.owasp.org/index.php/OWASP_Guide_Project`, 2015. [Online; Zugriff 13.11.2015].

[Ope15c] Open Web Application Security Project. OWASP Testing Project. `https://www.owasp.org/index.php/OWASP_Testing_Project`, 2015. [Online; Zugriff 13.11.2015].

[Ope15d] Open Web Application Security Project. OWASP Zed Attack Proxy. `https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project`, 2015. [Online; Zugriff 13.11.2015].

[Ope16] OpenVAS Community. OpenVAS. `http://www.openvas.org/`, 2016. [Online; Zugriff 13.02.2016].

[Paf15] Markus Paff. Entwicklung einer Security-Architektur-Beschreibungssprache. Masterarbeit, Rheinisch Westfälische Technische Hochschule Aachen, 2015.

[PAS02] Calvin S. Powers, Paul Ashley, and Matthias Schunter. Privacy Promises, Access Control, and Privacy Management – Enforcing Privacy Throughout an Enterprise By Extending Access Control. In *International Symposium on Electronic Commerce*, pages 13–21. IEEE, 2002.

[PC10] Shari L. Pfleeger and Robert K. Cunningham. Why Measuring Security Is Hard. *IEEE Security & Privacy*, 8(4):46–54, July 2010.

[Pea09] Siani Pearson. Taking Account of Privacy when Designing Cloud Computing Services. In *ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 44–52. IEEE, 2009.

[PM03] Andreas Pashalidis and Chris J. Mitchell. A Taxonomy of Single Sign-On Systems. In Rei Safavi-Naini and Jennifer Seberry, editors, *Information Security and Privacy*, volume 2727 of *LNCS*, pages 249–264. Springer, 2003.

[Por15] Portswigger. Burp. `https://portswigger.net/burp/`, 2015. [Online; Zugriff 13.11.2015].

[Pos81] J. Postel. RFC793: Transmission Control Protocol, 1981.

[Pot15] Lucian Poth. Generierung von Infrastruktur-Code für die Authentifizierung und Autorisierung aus MontiSecArc-Modellen. Bachelorarbeit, Rheinisch Westfälische Technische Hochschule Aachen, 2015.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM)*, volume 1708 of *LNCS*, pages 96–115. Springer, 1999.

[Pro15] Projekthaus HumTec. IPACS. `http://www.humtec.rwth-aachen.de/index.php?article_id=1034`, 2015. [Online; Zugriff 13.11.2015].

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[R C14] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[Rai15] Philipp Rainisch. Using MontiSecArc for modeling Physical Tamper Resistance. Bachelorarbeit, Rheinisch Westfälische Technische Hochschule Aachen, 2015.

[RO10] Tillmann Rendel and Klaus Ostermann. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing. In *ACM Haskell Symposium on Haskell*, pages 1–12. ACM, 2010.

[Roe99] Martin Roesch. Snort: Lightweight Intrusion Detection for Networks. In *USENIX Conference on Systems Administration (LISA)*, pages 229–238. USENIX Association, 1999.

[Rom01] Sasha Romanosky. Security Design Patterns Part 1. 2001.

[Ros12] Jeffrey Rosen. The Right to Be Forgotten. *Stanford Law Review Online*, 64:88–92, 2012.

[RR13] Dirk Reiss and Bernhard Rumpe. Using Lightweight Activity Diagrams for Modeling and Generation of Web Information Systems. In *Information Systems: Methods, Models, and Applications – International United Information Systems Conference (UNISCON)*, volume 137 of *Lecture Notes in Business Information Processing*, pages 61–73. Springer, 2013.

[RT06] Jie Ren and Richard Taylor. A Secure Software Architecture Description Language. In Elizabeth Fong, editor, *Workshop on Software Security Assurance Tools, Techniques, and Metrics, Special Publication (SP) 500-265*. National Institute of Standards and Technology (NIST), February 2006.

[RTDR05] Jie Ren, Richard Taylor, Paul Dourish, and David Redmiles. Towards an Architectural Treatment of Software Security: A Connector-centric Approach. *SIGSOFT Software Engineering Notes*, 30(4):1–7, May 2005.

[Rum11] Bernhard Rumpe. *Modellierung mit UML.* Springer Berlin, 2te edition, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. Springer Berlin, 2te edition, Juni 2012.

[SC09] Sarah Spiekermann and Lorrie F. Cranor. Engineering Privacy. *IEEE Transactions on Software Engineering*, 35(1):67–82, January 2009.

[Sch99] Bruce Schneier. Attack Trees. *Dr. Dobb's journal*, 24(12):21–29, 1999.

[Sch04] Thomas Schreiber. Session Riding: A Widespread Vulnerability in Today's Web Applications. 2004.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Number 11 in Aachener Informatik-Berichte, Software Engineering. Shaker Verlag, 2012.

[Sea13] Robert C. Seacord. *Secure Coding in C and C++*. SEI Series in Software Engineering. Addison-Wesley Professional, 2nd edition, 2013.

[SEH13] Teodor Sommestad, Matthias Ekstedt, and Hannes Holm. The Cyber Security Modeling Language: A Tool for Assessing the Vulnerability of Enterprise System Architectures. *IEEE Systems Journal*, 7(3):363–373, 2013.

[Sel03] Bran Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[SFBH+13] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns*. Wiley, 2013.

[SG12] Craig Silverstein and Google Inc. How To Use the Ctemplate (formerly Google Template) System. `https://google-ctemplate.googlecode.com/svn/trunk/doc/guide.html`, 2012. [Online; Zugriff 13.11.2015].

[SGS12] Ina Schieferdecker, Juergen Grossmann, and Martin Schneider. Model-Based Security Testing. In Alexander K. Petrenko and Holger Schlingloff, editors, *Workshop on Model-Based Testing*, volume 80 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–12. Open Publishing Association, 2012.

[Sho14] Adam Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[Sil11] Raúl Siles. SAP: Session (Fixation) Attacks and Protections (in Web Applications). In *Black Hat Europe*, 2011.

[Sko05] Sergei P. Skorobogatov. Semi-invasive attacks – A new approach to hardware security analysis. Technical Report UCAM-CL-TR-630, University of Cambridge, Computer Laboratory, 2005.

[SL15] James Sullivan and Michael E. Locasto. Defining a Model for Defense-In-Depth. In *Layered Assurance Workshop (LAW)*, 2015.

[Smi12] Ian G. Smith, editor. *The Internet of Things 2012 – New Horizons*. Internet of Things European Research Cluster (IERC), 2012.

[SML11] Prateek Saxena, David Molnar, and Benjamin Livshits. SCRIPTGARD: Automatic Context-sensitive Sanitization for Large-scale Legacy Web Applications. In *Conference on Computer and Communications Security (CCS)*, pages 601–614. ACM, 2011.

[SNO06] Fumiko Satoh, Yuichi. Nakamura, and Koichi Ono. Adding Authentication to Model Driven Security. In *International Conference on Web Services (ICWS)*, pages 585–594, September 2006.

[Sof15] Software Engineering Institute. OSATE2. `https://wiki.sei.cmu.edu/aadl/index.php/Osate_2`, 2015. [Online; Zugriff 13.11.2015].

[SPBS11] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Anna Shubina. The Halting problems of network stack insecurity. *;login*, 36(6):22–32, 2011.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[SS94] Ravi S. Sandhu and Pierangela Samarati. Access Control: Principle and Practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[SS13] Bjørnar Solhaug and Ketil Stølen. The CORAS Language-Why it is designed the way it is. In *International Conference on Structural Safety and Reliability (ICOSSAR)*, pages 3155–3162, 2013.

[SSS11] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive Auto-sanitization in Web Templating Languages Using Type Qualifiers. In *ACM Conference on Computer and Communications Security (CCS)*, pages 587–600. ACM, 2011.

[Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer Wien, 1973.

[The15] The jQuery Foundation. jQuery. `https://jquery.com`, 2015. [Online; Zugriff 13.11.2015].

[TJA10] Hassan Takabi, James B.D. Joshi, and Gail-Joon Ahn. Security and Privacy Challenges in Cloud Computing Environments. *IEEE Security Privacy*, 8(6):24–31, 2010.

[TMD09] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[TSFMP09] Juan Trujillo, Emilio Soler, Eduardo Fernández-Medina, and Mario Piattini. An engineering process for developing Secure Data Warehouses. *Information and Software Technology*, 51(6):1033 – 1051, 2009.

[TSR11] Slim Trabelsi, Jakub Sendor, and Stefanie Reinicke. PPL: PrimeLife Privacy Policy Engine. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, pages 184–185. IEEE, 2011.

[VBFMM12] Belén Vela, Carlos Blanco, Eduardo Fernández-Medina, and Esperanza Marcos. A practical application of our MDD approach for modeling secure XML data warehouses. *Decision Support Systems*, 52(4):899 – 925, 2012.

[vdBSYJ15] Alexander van den Berghe, Riccardo Scandariato, Koen Yskout, and Wouter Joosen. Design notations for secure software: a systematic literature review. *Software & Systems Modeling (SoSyM)*, pages 1–23, 2015.

[vdBV96] Mark van den Brand and Eelco Visser. Generation of Formatters for Context-free Languages. *ACM Transactions Software Engineering Methodology (TOSEM)*, 5(1):1–41, January 1996.

[vdSO] Andrew van der Stock and OWASP Community. Owasp Guide Project. `https://www.owasp.org/index.php/OWASP_Guide_Project`.

[Vis02] Eelco Visser. Meta-programming with Concrete Object Syntax. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *LNCS*, pages 299–315. Springer, 2002.

[VM02] John Viega and Gary McGraw. *Building Secure Software*. Addison-Wesley, 2002.

[Wad98] Philip Wadler. A Prettier Printer. In *Journal of Functional Programming*, pages 223–244. Palgrave Macmillan, 1998.

[Wat01] Peter Watkins. Cross-Site Request Forgeries (Re: The Dangers of Allowing Users to Post Images). *Bugtraq Mailing List*, 2001.

[Wei00] Steve H. Weingart. Physical Security Devices for Computer Subsystems: A Survey of Attacks and Defenses. In Çetin K. Koç and Christof Paar, editors, *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 302–317. Springer, 2000.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Number 12 in Aachener Informatik-Berichte, Software Engineering. Shaker Verlag, 2012.

[Win03] Janette M. Wing. A Call to Action: Look Beyond the Horizon. *IEEE Security & Privacy*, 99(6):62–67, November 2003.

[WO13] Dave Wichers and OWASP Community. OWASP Top Ten Project. https://www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013, 2013. [Online; Zugriff 13.11.2015].

[WSA+11] Joel Weinberger, Prateek Saxena, Devdatta Akhawe, Matthew Finifter, Richard Shin, and Dawn Song. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In *Computer Security – ESORICS 2011*, volume 6879 of *LNCS*, pages 150–171. Springer, 2011.

[WZ75] Gregory L. White and Philip G. Zimbardo. The Chilling Effects of Surveillance: Deindividuation and Reactance. Technical Report Z-15, National Technical Information Service, U. S. Department of Commerce, 1975.

[YB97] Joseph Yoder and Jeffrey Barcalow. Architectural Patterns for Enabling Application Security. In *Pattern Languages of Programming Conference (PLoP)*, 1997.

[YHDM04] Huiqun Yu, Xudong He, Yi Deng, and Lian Mo. A Formal Approach to Designing Secure Software Architectures. In *IEEE International Symposium on High Assurance Systems Engineering*, pages 289–290. IEEE, March 2004.

[YHSJ06] Koen Yskout, Thomas Heyman, Riccardo Scandariato, and Wouter Joosen. A System of Security Patterns. Technical Report CW 469, Department of Computer Science, Katholieke Universiteit Leuven, 2006.

[YWM08] Nobukazu Yoshioka, Hironori Washizaki, and Katsuhisa Maruyama. A Survey on Security Patterns. *Progress in Informatics*, 5(5):35–47, 2008.

[ZAF08] Jie Zhou and Jim Alves-Foss. Security Policy Refinement and Enforcement for the Design of Multi-level Secure Systems. *Journal of Computer Security*, 16(2):107–131, April 2008.

[ZB14] Vadim Zaytsev and Anya H. Bagge. Parsing in a Broad Sense. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems (MODELS)*, volume 8767 of *LNCS*, pages 50–67. Springer, 2014.

[ZGMW14] Jan Henrik Ziegeldorf, Oscar Garcia Morchon, and Klaus Wehrle. Privacy in the Internet of Things: Threats and Challenges. *Security and Communication Networks*, 7(12):2728–2742, 2014.

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Definitionen

Anhang A.

Grammatiken

```
package secarc;

version "0.0.3-SNAPSHOT";

/**
 * This grammar modifies and extends the common
 * ArchitectureDiagram grammar
 * by adding security-specific concepts.
 *
 * Derived from mc.umlp.arcd.MontiArc
 */
grammar MontiSecArc extends mc.umlp.arc.MontiArc {

  options {
    compilationunit ArcComponent
    nostring
  }

  /**
   * SecArcTrustLevel defines the trust in a component.
   *
   * {@attribute stereotype an optional stereotype}
   * {@attribute value an absolute trust level}
   * {@attribute reason an optional reason for the trust level}
   */
  /SecArcTrustLevel implements
    (Stereotype? "trustlevel") => ArcElement =
    Stereotype?
    "trustlevel" ( ["+"] | ["-"] ) value:IntLiteral reason:String?
  ";";
```

MCG

Listing A.1: MontiSecArc Grammatik

MCG

```
/**
 * SecArcIdentity defines an Identity Link
 * between two components.
 */
/SecArcIdentity implements
  (Stereotype? "identity") => ArcElement =
  Stereotype?
  "identity" ( ["strong"] | ["weak"] ) source:QualifiedName "->"
  targets:QualifiedName ("," targets:QualifiedName)*
";";

/**
 * SecArcConnector represents an encrypted
 * connection between a source port
 * and a target port. It extends ArcConnector by
 * adding a keyword "encrypted".
 */
/SecArcConnector
  extends ArcConnector
  implements
  (Stereotype? "connect"
    ("encrypted"|"unencrypted")
    QualifiedName "->"
  )  => ArcElement =
  Stereotype? "connect"
  ( ["encrypted"] | ["unencrypted"] )?
  source:QualifiedName "->"
  targets:QualifiedName ("," targets:QualifiedName)*
";";

abstract ArcConnector;

/**
 * Simple way to connect ports encrypted.
 */
/SecArcSimpleConnector extends ArcSimpleConnector=
  Stereotype? ( ["encrypted"] | ["unencrypted"] )?
  source:QualifiedName "->"
  targets:QualifiedName
  ("," targets:QualifiedName)*
;

abstract ArcSimpleConnector;

/**
 * MontiSecArcAutoConnect is used to connect ports
 * automatically encrypted.
 */
/MontiSecArcAutoConnect extends MontiArcAutoConnect =
  "autoconnect" Stereotype? ( ["encrypted"] | ["unencrypted"] )?
  (["type"] | ["port"] | ["off"])
";";

abstract MontiArcAutoConnect;
```

Listing A.2: MontiSecArc Grammatik

```
85  /**
86   * The component of MontiSecArc.
87   */
88  /SecArcComponent extends ArcComponent
89    implements (Stereotype? "component"
90    Name Name? ArcComponentHead "{")  => ArcElement =
91    Stereotype?
92    "component" Name (instanceName:Name)?
93    head:ArcComponentHead
94    body:ArcComponentBody
95  ;
96
97  abstract ArcComponent;
98
99  /**
100  * The subcomponent of MontiSecArc.
101  */
102 /SecArcSubComponent extends ArcSubComponent
103   implements (Stereotype? "component" ReferenceType
104   ("(" | Name | ";" ))  => ArcElement =
105   Stereotype?
106   "component"
107   type:ReferenceType
108   ("(" arguments:CVExpression
109   ("," arguments:CVExpression)* ")" )?
110   (instances:ArcSubComponentInstance
111   ("," instances:ArcSubComponentInstance)* )?
112 ";";
113
114 abstract ArcSubComponent;
115
116 /**
117  * SecArcPort represents a port declaration.
118  * It extends ArcPort by adding a
119  * key word "critical" and the optional
120  * declaration of a critical port.
121  */
122 /SecArcPort
123   extends (Stereotype? ("critical")? ("in" | "out")) =>
124   ArcPort =
125   Stereotype?
126   (["critical"])?
127   (incoming:["in"] | outgoing:["out"])
128   Type Name?
129 ;
130
131 /**
132  * SecArcRole represents a set of roles for components or ports.
133  * A component/role has a selection of roles which have access.
134  * Default is public.
135  */
136 /SecArcRole
137   implements (Stereotype? "access" Name) => ArcElement =
138   Stereotype?
```

Listing A.3: MontiSecArc Grammatik

```
139     "access" roles:Name ("," roles:Name)*
140   ";";
141
142   /**
143    * SecArcRefRole represents a set of roles for ports or
144    * components. A port/component has a selection of roles
145    * which have access. You can reference the port/component.
146    */
147   /SecArcRefRole =
148     Stereotype?
149     portName:Name "(" roles:Name ("," roles:Name)* ")"
150   ;
151
152   /**
153    * SecArcRoleList represents a set of roles for a list of ports
154    * or components. A port/component
155    * has a selection of roles which have access.
156    */
157   /SecArcRoleInterface
158     implements (Stereotype? "access" QualifiedName "(" Name) =>
159     ArcElement =
160     Stereotype?
161     "access" SecArcRefRole ("," SecArcRefRole)*
162   ";";
163
164   /**
165    * SecArcPEP is a super-class of roles which is more common.
166    * Somewhere
167    * in the given component must be policy enforced with roles.
168    * Default is off.
169    */
170   /SecArcPEP
171     implements (Stereotype? "accesscontrol") => ArcElement =
172     Stereotype?
173     "accesscontrol" ( ["on"] | ["off"] )
174   ";";
175
176   /**
177    * SecArcCPE represent the cpe of a 3rd party component.
178    */
179   /SecArcCPE
180     implements(Stereotype? "cpe") => ArcElement =
181     Stereotype?
182     "cpe" nameCPE:StringLiteral
183   ";";
184
185   /**
186    * SecArcConfigration represent the configuration of a
187    * 3rd party component.
188    */
189   /SecArcConfiguration
190     implements(Stereotype? "configuration" Name) => ArcElement =
191     Stereotype?
192     "configuration" name:Name
193   ";";
```

MCG

Listing A.4: MontiSecArc Grammatik

```
194   /**
195    * SecArcTrustlevelRelation defines the trustlevel relation
196    * between two components.
197    */
198   /SecArcTrustlevelRelation
199     implements(Stereotype? "trustlevelrelation" QualifiedName) =>
200     ArcElement =
201     Stereotype?
202     "trustlevelrelation" client:QualifiedName
203     (
204         [">"]
205       |
206         ["="]
207       |
208         ["<"]
209     )
210     server:QualifiedName
211   ";";
212
213   // replacement of ASTCNode with UMLPNode
214   ast SecArcTrustLevel astextends
215       /mc.umlp.common._ast.UMLPNode;
216   ast SecArcIdentity astextends
217       /mc.umlp.common._ast.UMLPNode;
218   ast SecArcRole astextends
219       /mc.umlp.common._ast.UMLPNode;
220   ast SecArcRefRole astextends
221       /mc.umlp.common._ast.UMLPNode;
222   ast SecArcRoleInterface astextends
223       /mc.umlp.common._ast.UMLPNode;
224   ast SecArcPEP astextends
225       /mc.umlp.common._ast.UMLPNode;
226   ast SecArcCPE astextends
227       /mc.umlp.common._ast.UMLPNode;
228   ast SecArcConfiguration astextends
229       /mc.umlp.common._ast.UMLPNode;
230   ast SecArcTrustlevelRelation astextends
231       /mc.umlp.common._ast.UMLPNode;
232 }
```

MCG

Listing A.5: MontiSecArc Grammatik

MCG

```
Grammar =
  "grammar" Name
    ("extends" supergrammar:GrammarReference ("," supergrammar:
       GrammarReference)* )?
    "{"
       (
         GrammarOption
       |
         LexProd
       |
         EncodeTableProd
       |
         ClassProd
       |
         EnumProd
       |
         ExternalProd
       |
         InterfaceProd
       |
         AbstractProd
       |
         ASTRule
       |
         AssociationBlock
       |
         Association
       |
         Concept
       )*
    "}" ;

[...]

  EncodeTableProd implements Prod =
    "encodeTable" Name
    "="
    "{"
      ("options" "{"
        (encodeTableOptions:EncodeTableOption ";")*
      "}")?
      EncodeTableEntries:EncodeTableEntry
      ("," EncodeTableEntries:EncodeTableEntry)*
    "}"
    ";";
```

Listing A.6: MontiCoder Erweiterung in der MontiCore Grammatik Grammatik.

```
45  EncodeTableEntry =                                                    MCG
46    (
47      char:CHAR
48      |
49      string:STRING
50    )
51    "->"
52    replacement:STRING
53    (
54      "options" "{"
55        (encodeTableEntryOptions:EncodeTableEntryOption ";")*
56      "}"
57    )?
58  ;
59
60  EncodeTableEntryOption =
61    ["caseInsensitive"] | ["ignoreWS"]
62  ;
63
64  interface EncodeTableOption;
65
66  EncodeTableOptionPrefix implements EncodeTableOption =
67    "prefix" "=" prefix:STRING
68  ;
69
70  EncodeTableOptionSuffix implements EncodeTableOption =
71    "suffix" "=" suffix:STRING
72  ;
```

Listing A.7: MontiCoder Erweiterung in der MontiCore Grammatik Grammatik.

Anhang B.

Lebenslauf

Name			Hermerschmidt
Vorname			Lars
Geburtsort			Burgwedel
Geburtstag			1.4.1982

5/2016	-		Information Security Officer bei der AXA Konzern AG
1/2012	-	4/2016	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Softwareengineering der RWTH Aachen
10/2008	-	12/2011	Security Consultant bei der T-Systems International GmbH
10/2002	-	5/2008	Studium der Informatik an der RWTH Aachen Abschluss: Diplom mit Auszeichnung
7/2001	-	4/2002	Zivildienst
1998	-	2001	BBS Burgdorf Lehrte Fachgymnasium Technik Abschluss: Allgemeine Hochschulreife
1988	-	1998	Grundschule Wettmar, Orientierungsstufe Großburgwedel, Realschule Großburgwedel

Related Interesting Work from the SE Group, RWTH Aachen

Agile Model Based Software Engineering

Agility and modeling in the same project? This question was raised in [Rum04]: "Using an executable, yet abstract and multi-view modeling language for modeling, designing and programming still allows to use an agile development process." Modeling will be used in development projects much more, if the benefits become evident early, e.g with executable UML [Rum02] and tests [Rum03]. In [GKRS06], for example, we concentrate on the integration of models and ordinary programming code. In [Rum12] and [Rum16], the UML/P, a variant of the UML especially designed for programming, refactoring and evolution, is defined. The language workbench MontiCore [GKR+06, GKR+08] is used to realize the UML/P [Sch12]. Links to further research, e.g., include a general discussion of how to manage and evolve models [LRSS10], a precise definition for model composition as well as model languages [HKR+09] and refactoring in various modeling and programming languages [PR03]. In [FHR08] we describe a set of general requirements for model quality. Finally [KRV06] discusses the additional roles and activities necessary in a DSL-based software development project. In [CEG+14] we discuss how to improve reliability of adaptivity through models at runtime, which will allow developers to delay design decisions to runtime adaptation.

Generative Software Engineering

The UML/P language family [Rum12, Rum11, Rum16] is a simplified and semantically sound derivative of the UML designed for product and test code generation. [Sch12] describes a flexible generator for the UML/P based on the MontiCore language workbench [KRV10, GKR+06, GKR+08]. In [KRV06], we discuss additional roles necessary in a model-based software development project. In [GKRS06] we discuss mechanisms to keep generated and handwritten code separated. In [Wei12] demonstrate how to systematically derive a transformation language in concrete syntax. To understand the implications of executability for UML, we discuss needs and advantages of executable modeling with UML in agile projects in [Rum04], how to apply UML for testing in [Rum03] and the advantages and perils of using modeling languages for programming in [Rum02].

Unified Modeling Language (UML)

Starting with an early identification of challenges for the standardization of the UML in [KER99] many of our contributions build on the UML/P variant, which is described in the two books [Rum16] and [Rum12] implemented in [Sch12]. Semantic variation points of the UML are discussed in [GR11]. We discuss formal semantics for UML [BHP+98] and describe UML semantics using the "System Model" [BCGR09a], [BCGR09b], [BCR07b] and [BCR07a]. Semantic variation points have, e.g., been applied to define class diagram semantics [CGR08]. A precisely defined semantics for variations is applied, when checking variants of class diagrams [MRR11c] and objects diagrams [MRR11d] or the consistency of both kinds of diagrams [MRR11e]. We also apply these concepts to activity diagrams [MRR11b] which allows us to check for semantic differences of activity diagrams [MRR11a]. The basic semantics for ADs and their semantic variation points is given in [GRR10]. We also discuss how to ensure and identify model quality [FHR08], how models, views and the system under development correlate to each other [BGH+98] and how to use modeling in agile development projects [Rum04], [Rum02]. The question how to adapt and extend the UML is discussed in [PFR02] describing product line annotations for UML and more general discussions and insights on how to use meta-modeling for defining and adapting the UML are included in [EFLR99], [FELR98] and [SRVK10].

Domain Specific Languages (DSLs)

Computer science is about languages. Domain Specific Languages (DSLs) are better to use, but need appropriate tooling. The MontiCore language workbench [GKR+06, KRV10, Kra10, GKR+08] allows the specification of an integrated abstract and concrete syntax format [KRV07b] for easy development. New languages and tools can be defined in modular forms [KRV08, GKR+07, Völ11] and can, thus, easily be reused. [Wei12] presents a tool that allows to create transformation rules tailored to an underlying DSL. Variability in DSL definitions has been examined in [GR11]. A successful application has been carried out in the Air Traffic Management domain [ZPK+11]. Based on the concepts described above, meta modeling, model analyses and model evolution have been discussed in [LRSS10] and [SRVK10]. DSL quality [FHR08], instructions for defining views [GHK+07], guidelines to define DSLs [KKP+09] and Eclipse-based tooling for DSLs [KRV07a] complete the collection.

Software Language Engineering

For a systematic definition of languages using composition of reusable and adaptable language components, we adopt an engineering viewpoint on these techniques. General ideas on how to engineer a language can be found in the GeMoC initiative [CBCR15, CCF+15]. As said, the MontiCore language workbench provides techniques for an integrated definition of languages [KRV07b, Kra10, KRV10]. In [SRVK10] we discuss the possibilities and the challenges using metamodels for language definition. Modular composition, however, is a core concept to reuse language components like in MontiCore for the frontend [Völ11, KRV08] and the backend [RRRW15]]. Language derivation is to our believe a promising technique to develop new languages for a specific purpose that rely on existing basic languages. How to automatically derive such a transformation language using concrete syntax of the base language is described in [HRW15, Wei12] and successfully applied to various DSLs. We also applied the language derivation technique to tagging languages that decorate a base language [GLRR15] and delta languages [HHK+15a, HHK+13], where a delta language is derived from a base language to be able to constructively describe differences between model variants usable to build feature sets.

Modeling Software Architecture & the MontiArc Tool

Distributed interactive systems communicate via messages on a bus, discrete event signals, streams of telephone or video data, method invocation, or data structures passed between software services. We use streams, statemachines and components [BR07] as well as expressive forms of composition and refinement [PR99] for semantics. Furthermore, we built a concrete tooling infrastructure called MontiArc [HRR12] for architecture design and extensions for states [RRW13b]. MontiArc was extended to describe variability [HRR+11] using deltas [HRRS11, **?**] and evolution on deltas [HRRS12]. [GHK+07] and [GHK+08] close the gap between the requirements and the logical architecture and [GKPR08] extends it to model variants. [MRR14] provides a precise technique to verify consistency of architectural views [Rin14, MRR13] against a complete architecture in order to increase reusability. Co-evolution of architecture is discussed in [MMR10] and a modeling technique to describe dynamic architectures is shown in [HRR98].

Compositionality & Modularity of Models

[HKR+09] motivates the basic mechanisms for modularity and compositionality for modeling. The mechanisms for distributed systems are shown in [BR07] and algebraically underpinned in [HKR+07]. Semantic and methodical aspects of model composition [KRV08] led to the language workbench MontiCore [KRV10] that can even be used to develop modeling tools in a compositional form. A set of DSL design guidelines incorporates reuse through this form of composition [KKP+09]. [Völ11] examines the composition of context conditions respectively the underlying infrastructure of the symbol table. Modular

editor generation is discussed in [KRV07a]. [RRRW15] applies compositionality to Robotics control. [CBCR15] (published in [CCF+15]) summarizes our approach to composition and remaining challenges in form of a conceptual model of the "globalized" use of DSLs. As a new form of decomposition of model information we have developed the concept of tagging languages in [GLRR15]. It allows to describe additional information for model elements in separated documents, facilitates reuse, and allows to type tags.

Semantics of Modeling Languages

The meaning of semantics and its principles like underspecification, language precision and detailedness is discussed in [HR04]. We defined a semantic domain called "System Model" by using mathematical theory in [RKB95, BHP+98] and [GKR96, KRB96]. An extended version especially suited for the UML is given in [BCGR09b] and in [BCGR09a] its rationale is discussed. [BCR07a, BCR07b] contain detailed versions that are applied to class diagrams in [CGR08]. To better understand the effect of an evolved design, detection of semantic differencing as opposed to pure syntactical differences is needed [MRR10]. [MRR11a, MRR11b] encode a part of the semantics to handle semantic differences of activity diagrams and [MRR11e] compares class and object diagrams with regard to their semantics. In [BR07], a simplified mathematical model for distributed systems based on black-box behaviors of components is defined. Meta-modeling semantics is discussed in [EFLR99]. [BGH+97] discusses potential modeling languages for the description of an exemplary object interaction, today called sequence diagram. [BGH+98] discusses the relationships between a system, a view and a complete model in the context of the UML. [GR11] and [CGR09] discuss general requirements for a framework to describe semantic and syntactic variations of a modeling language. We apply these on class and object diagrams in [MRR11e] as well as activity diagrams in [GRR10]. [Rum12] defines the semantics in a variety of code and test case generation, refactoring and evolution techniques. [LRSS10] discusses evolution and related issues in greater detail.

Evolution & Transformation of Models

Models are the central artifact in model driven development, but as code they are not initially correct and need to be changed, evolved and maintained over time. Model transformation is therefore essential to effectively deal with models. Many concrete model transformation problems are discussed: evolution [LRSS10, MMR10, Rum04], refinement [PR99, KPR97, PR94], refactoring [Rum12, PR03], translating models from one language into another [MRR11c, Rum12] and systematic model transformation language development [Wei12]. [Rum04] describes how comprehensible sets of such transformations support software development and maintenance [LRSS10], technologies for evolving models within a language and across languages, and mapping architecture descriptions to their implementation [MMR10]. Automaton refinement is discussed in [PR94, KPR97], refining pipe-and-filter architectures is explained in [PR99]. Refactorings of models are important for model driven engineering as discussed in [PR01, PR03, Rum12]. Translation between languages, e.g., from class diagrams into Alloy [MRR11c] allows for comparing class diagrams on a semantic level.

Variability & Software Product Lines (SPL)

Products often exist in various variants, for example cars or mobile phones, where one manufacturer develops several products with many similarities but also many variations. Variants are managed in a Software Product Line (SPL) that captures product commonalities as well as differences. Feature diagrams describe variability in a top down fashion, e.g., in the automotive domain [GHK+08] using 150% models. Reducing overhead and associated costs is discussed in [GRJA12]. Delta modeling is a bottom up technique starting with a small, but complete base variant. Features are additive, but also can modify the

core. A set of commonly applicable deltas configures a system variant. We discuss the application of this technique to Delta-MontiArc [HRR+11, HRR+11] and to Delta-Simulink [HKM+13]. Deltas can not only describe spacial variability but also temporal variability which allows for using them for software product line evolution [HRRS12]. [HHK+13] and [HRW15] describe an approach to systematically derive delta languages. We also apply variability to modeling languages in order to describe syntactic and semantic variation points, e.g., in UML for frameworks [PFR02]. Furthermore, we specified a systematic way to define variants of modeling languages [CGR09] and applied this as a semantic language refinement on Statecharts in [GR11].

Cyber-Physical Systems (CPS)

Cyber-Physical Systems (CPS) [KRS12] are complex, distributed systems which control physical entities. Contributions for individual aspects range from requirements [GRJA12], complete product lines [HRRW12], the improvement of engineering for distributed automotive systems [HRR12] and autonomous driving [BR12a] to processes and tools to improve the development as well as the product itself [BBR07]. In the aviation domain, a modeling language for uncertainty and safety events was developed, which is of interest for the European airspace [ZPK+11]. A component and connector architecture description language suitable for the specific challenges in robotics is discussed in [RRW13b, RRW14]. Monitoring for smart and energy efficient buildings is developed as Energy Navigator toolset [KPR12, FPPR12, KLPR12].

State Based Modeling (Automata)

Today, many computer science theories are based on statemachines in various forms including Petri nets or temporal logics. Software engineering is particularly interested in using statemachines for modeling systems. Our contributions to state based modeling can currently be split into three parts: (1) understanding how to model object-oriented and distributed software using statemachines resp. Statecharts [GKR96, BCR07b, BCGR09b, BCGR09a], (2) understanding the refinement [PR94, RK96, Rum96] and composition [GR95] of statemachines, and (3) applying statemachines for modeling systems. In [Rum96] constructive transformation rules for refining automata behavior are given and proven correct. This theory is applied to features in [KPR97]. Statemachines are embedded in the composition and behavioral specification concepts of Focus [BR07]. We apply these techniques, e.g., in MontiArcAutomaton [RRW13a, RRW14] as well as in building management systems [FLP+11].

Robotics

Robotics can be considered a special field within Cyber-Physical Systems which is defined by an inherent heterogeneity of involved domains, relevant platforms, and challenges. The engineering of robotics applications requires composition and interaction of diverse distributed software modules. This usually leads to complex monolithic software solutions hardly reusable, maintainable, and comprehensible, which hampers broad propagation of robotics applications. The MontiArcAutomaton language [RRW13a] extends ADL MontiArc and integrates various implemented behavior modeling languages using MontiCore [RRW13b, RRW14, RRRW15] that perfectly fit Robotic architectural modeling. The LightRocks [THR+13] framework allows robotics experts and laymen to model robotic assembly tasks.

Automotive, Autonomic Driving & Intelligent Driver Assistance

Introducing and connecting sophisticated driver assistance, infotainment and communication systems as well as advanced active and passive safety-systems result in complex embedded systems. As these feature-driven subsystems may be arbitrarily combined by the customer, a huge amount of distinct variants needs

to be managed, developed and tested. A consistent requirements management that connects requirements with features in all phases of the development for the automotive domain is described in [GRJA12]. The conceptual gap between requirements and the logical architecture of a car is closed in [GHK+07, GHK+08]. [HKM+13] describes a tool for delta modeling for Simulink [HKM+13]. [HRRW12] discusses means to extract a well-defined Software Product Line from a set of copy and paste variants. [RSW+15] describes an approach to use model checking techniques to identify behavioral differences of Simulink models. Quality assurance, especially of safety-related functions, is a highly important task. In the Carolo project [BR12a, BR12b], we developed a rigorous test infrastructure for intelligent, sensor-based functions through fully-automatic simulation [BBR07]. This technique allows a dramatic speedup in development and evolution of autonomous car functionality, and thus enables us to develop software in an agile way [BR12a]. [MMR10] gives an overview of the current state-of-the-art in development and evolution on a more general level by considering any kind of critical system that relies on architectural descriptions. As tooling infrastructure, the SSELab storage, versioning and management services [HKR12] are essential for many projects.

Energy Management

In the past years, it became more and more evident that saving energy and reducing CO2 emissions is an important challenge. Thus, energy management in buildings as well as in neighborhoods becomes equally important to efficiently use the generated energy. Within several research projects, we developed methodologies and solutions for integrating heterogeneous systems at different scales. During the design phase, the Energy Navigators Active Functional Specification (AFS) [FPPR12, KPR12] is used for technical specification of building services already. We adapted the well-known concept of statemachines to be able to describe different states of a facility and to validate it against the monitored values [FLP+11]. We show how our data model, the constraint rules and the evaluation approach to compare sensor data can be applied [KLPR12].

Cloud Computing & Enterprise Information Systems

The paradigm of Cloud Computing is arising out of a convergence of existing technologies for web-based application and service architectures with high complexity, criticality and new application domains. It promises to enable new business models, to lower the barrier for web-based innovations and to increase the efficiency and cost-effectiveness of web development [KRR14]. Application classes like Cyber-Physical Systems and their privacy [HHK+14, HHK+15b], Big Data, App and Service Ecosystems bring attention to aspects like responsiveness, privacy and open platforms. Regardless of the application domain, developers of such systems are in need for robust methods and efficient, easy-to-use languages and tools [KRS12]. We tackle these challenges by perusing a model-based, generative approach [NPR13]. The core of this approach are different modeling languages that describe different aspects of a cloud-based system in a concise and technology-agnostic way. Software architecture and infrastructure models describe the system and its physical distribution on a large scale. We apply cloud technology for the services we develop, e.g., the SSELab [HKR12] and the Energy Navigator [FPPR12, KPR12] but also for our tool demonstrators and our own development platforms. New services, e.g., collecting data from temperature, cars etc. can now easily be developed.

]Literaturverzeichnis

[BBR07] Christian Basarke, Christian Berger, and Bernhard Rumpe. Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence. *Journal of Aerospace Computing, Information, and Communication (JACIC)*, 4(12):1158–1174, 2007.

[BCGR09a] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 43–61. John Wiley & Sons, November 2009.

[BCGR09b] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Definition of the UML System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–93. John Wiley & Sons, November 2009.

[BCR07a] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 2: The Control Model. Technical Report TUM-I0710, TU Munich, Germany, February 2007.

[BCR07b] Manfred Broy, María Victoria Cengarle, and Bernhard Rumpe. Towards a System Model for UML. Part 3: The State Machine Model. Technical Report TUM-I0711, TU Munich, Germany, February 2007.

[BGH+97] Ruth Breu, Radu Grosu, Christoph Hofmann, Franz Huber, Ingolf Krüger, Bernhard Rumpe, Monika Schmidt, and Wolfgang Schwerin. Exemplary and Complete Object Interaction Descriptions. In *Object-oriented Behavioral Semantics Workshop (OOPSLA'97)*, Technical Report TUM-I9737, Germany, 1997. TU Munich.

[BGH+98] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe, and Wolfgang Schwerin. Systems, Views and Models of UML. In *Proceedings of the Unified Modeling Language, Technical Aspects and Applications*, pages 93–109. Physica Verlag, Heidelberg, Germany, 1998.

[BHP+98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, and Katharina Spies. Software and System Modeling Based on a Unified Formal Semantics. In *Workshop on Requirements Targeting Software and Systems Engineering (RTSE'97)*, LNCS 1526, pages 43–68. Springer, 1998.

[BR07] Manfred Broy and Bernhard Rumpe. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik-Spektrum*, 30(1):3–18, Februar 2007.

[BR12a] Christian Berger and Bernhard Rumpe. Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System. In *Automotive Software Engineering Workshop (ASE'12)*, pages 789–798, 2012.

[BR12b] Christian Berger and Bernhard Rumpe. Engineering Autonomous Driving Software. In C. Rouff and M. Hinchey, editors, *Experience from the DARPA Urban Challenge*, pages 243–271. Springer, Germany, 2012.

[CBCR15] Tony Clark, Mark van den Brand, Benoit Combemale, and Bernhard Rumpe. Conceptual Model of the Globalization for Domain-Specific Languages. In *Globalizing Domain-Specific Languages*, LNCS 9400, pages 7–20. Springer, 2015.

[CCF+15] Betty H. C. Cheng, Benoit Combemale, Robert B. France, Jean-Marc Jézéquel, and Bernhard Rumpe, editors. *Globalizing Domain-Specific Languages*, LNCS 9400. Springer, 2015.

[CEG+14] Betty Cheng, Kerstin Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi Müller, Patrizio Pelliccione, Anna Perini, Nauman Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha Villegas. Using Models at Runtime to Address Assurance for Self-Adaptive Systems. In *Models@run.time*, LNCS 8378, pages 101–136. Springer, Germany, 2014.

[CGR08] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. System Model Semantics of Class Diagrams. Informatik-Bericht 2008-05, TU Braunschweig, Germany, 2008.

[CGR09] María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Variability within Modeling Language Definitions. In *Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, LNCS 5795, pages 670–684. Springer, 2009.

[EFLR99] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Meta-Modelling Semantics of UML. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 45–60. Kluver Academic Publisher, 1999.

[FELR98] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, November 1998.

[FHR08] Florian Fieber, Michaela Huhn, and Bernhard Rumpe. Modellqualität als Indikator für Softwarequalität: eine Taxonomie. *Informatik-Spektrum*, 31(5):408–424, Oktober 2008.

[FLP+11] M. Norbert Fisch, Markus Look, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. State-based Modeling of Buildings and Facilities. In *Enhanced Building Operations Conference (ICEBO'11)*, 2011.

[FPPR12] M. Norbert Fisch, Claas Pinkernell, Stefan Plesser, and Bernhard Rumpe. The Energy Navigator - A Web-Platform for Performance Design and Management. In *Energy Efficiency in Commercial Buildings Conference(IEECB'12)*, 2012.

[GHK+07] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, and Bernhard Rumpe. View-based Modeling of Function Nets. In *Object-oriented Modelling of Embedded Real-Time Systems Workshop (OMER4'07)*, 2007.

[GHK+08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt, and Bernhard Rumpe. Modelling Automotive Function Nets with Views for Features, Variants, and Modes. In *Proceedings of 4th European Congress ERTS - Embedded Real Time Software*, 2008.

[GKPR08] Hans Grönniger, Holger Krahn, Claas Pinkernell, and Bernhard Rumpe. Modeling Variants of Automotive Systems using Views. In *Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*, Informatik Bericht 2008-01, pages 76–89. TU Braunschweig, 2008.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab System Model with State. Technical Report TUM-I9631, TU Munich, Germany, July 1996.

[GKR+06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig, August 2006.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Textbased Modeling. In *4th International Workshop on Software Language Engineering, Nashville*, Informatik-Bericht 4/2007. Johannes-Gutenberg-Universität Mainz, 2007.

[GKR+08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: A Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926, 2008.

[GKRS06] Hans Grönniger, Holger Krahn, Bernhard Rumpe, and Martin Schindler. Integration von Modellen in einen codebasierten Softwareentwicklungsprozess. In *Modellierung 2006 Conference*, LNI 82, Seiten 67–81, 2006.

[GLRR15] Timo Greifenberg, Markus Look, Sebastian Roidl, and Bernhard Rumpe. Engineering Tagging Languages for DSLs. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 34–43. ACM/IEEE, 2015.

[GR95] Radu Grosu and Bernhard Rumpe. Concurrent Timed Port Automata. Technical Report TUM-I9533, TU Munich, Germany, October 1995.

[GR11] Hans Grönniger and Bernhard Rumpe. Modeling Language Variability. In *Workshop on Modeling, Development and Verification of Adaptive Systems*, LNCS 6662, pages 17–32. Springer, 2011.

[GRJA12] Tim Gülke, Bernhard Rumpe, Martin Jansen, and Joachim Axmann. High-Level Requirements Management and Complexity Costs in Automotive Development Projects: A Problem Statement. In *Requirements Engineering: Foundation for Software Quality (REFSQ'12)*, 2012.

[GRR10] Hans Grönniger, Dirk Reiß, and Bernhard Rumpe. Towards a Semantics of Activity Diagrams with Semantic Variation Points. In *Conference on Model Driven Engineering Languages and Systems (MODELS'10)*, LNCS 6394, pages 331–345. Springer, 2010.

[HHK+13] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, and Ina Schaefer. Engineering Delta Modeling Languages. In *Software Product Line Conference (SPLC'13)*, pages 22–31. ACM, 2013.

[HHK+14] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. User-driven Privacy Enforcement for Cloud-based Services in the Internet of Things. In *Conference on Future Internet of Things and Cloud (FiCloud'14)*. IEEE, 2014.

[HHK+15a] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. Systematic Synthesis of Delta Modeling Languages. *Journal on Software Tools for Technology Transfer (STTT)*, 17(5):601–626, October 2015.

[HHK+15b] Martin Henze, Lars Hermerschmidt, Daniel Kerpen, Roger Häußling, Bernhard Rumpe, and Klaus Wehrle. A comprehensive approach to privacy in the cloud-based Internet of Things. *Future Generation Computer Systems*, 56:701–718, 2015.

[HKM+13] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-Class Variability Modeling in Matlab/Simulink. In *Variability Modelling of Software-intensive Systems Workshop (VaMoS'13)*, pages 11–18. ACM, 2013.

[HKR+07] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. An Algebraic View on the Semantics of Model Composition. In *Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'07)*, LNCS 4530, pages 99–113. Springer, Germany, 2007.

[HKR+09] Christoph Herrmann, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Scaling-Up Model-Based-Development for Large Heterogeneous Systems with Compositional Modeling. In *Conference on Software Engineeering in Research and Practice (SERP'09)*, pages 172–176, July 2009.

[HKR+11] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *Software Architecture Conference (ECSA'11)*, pages 6:1–6:10. ACM, 2011.

[HKR12] Christoph Herrmann, Thomas Kurpick, and Bernhard Rumpe. SSELab: A Plug-In-Based Framework for Web-Based Project Portals. In *Developing Tools as Plug-Ins Workshop (TOPI'12)*, pages 61–66. IEEE, 2012.

[HR04] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, October 2004.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling Dynamic Component Interfaces. In *Technology of Object-Oriented Languages and Systems (TOOLS 26)*, pages 58–70. IEEE, 1998.

[HRR+11] Arne Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer, and Frank van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Lines Conference (SPLC'11)*, pages 150–159. IEEE, 2011.

[HRR12] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University, February 2012.

[HRRS11] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, pages 1 – 10. fortiss GmbH, 2011.

[HRRS12] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Evolving Delta-oriented Software Product Line Architectures. In *Large-Scale Complex IT Systems. Development, Operation and Management, 17th Monterey Workshop 2012*, LNCS 7539, pages 183–208. Springer, 2012.

[HRRW12] Christian Hopp, Holger Rendel, Bernhard Rumpe, and Fabian Wolf. Einführung eines Produktlinienansatzes in die automotive Softwareentwicklung am Beispiel von Steuergerätesoftware. In *Software Engineering Conference (SE'12)*, LNI 198, Seiten 181–192, 2012.

[HRW15] Katrin Hölldobler, Bernhard Rumpe, and Ingo Weisemöller. Systematically Deriving Domain-Specific Transformation Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'15)*, pages 136–145. ACM/IEEE, 2015.

[KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML Semantics FAQ. In A. Moreira and S. Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader*, LNCS 1743, Berlin, 1999. Springer Verlag.

[KKP+09] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design Guidelines for Domain Specific Languages. In *Domain-Specific Modeling Workshop (DSM'09)*, Techreport B-108, pages 7–13. Helsinki School of Economics, October 2009.

[KLPR12] Thomas Kurpick, Markus Look, Claas Pinkernell, and Bernhard Rumpe. Modeling Cyber-Physical Systems: Model-Driven Specification of Energy Efficient Buildings. In *Modelling of the Physical World Workshop (MOTPW'12)*, pages 2:1–2:6. ACM, October 2012.

[KPR97] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature Specification and Refinement with State Transition Diagrams. In *Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*, pages 284–297. IOS-Press, 1997.

[KPR12] Thomas Kurpick, Claas Pinkernell, and Bernhard Rumpe. Der Energie Navigator. In H. Lichter and B. Rumpe, Editoren, *Entwicklung und Evolution von Forschungssoftware. Tagungsband, Rolduc, 10.-11.11.2011*, Aachener Informatik-Berichte, Software Engineering, Band 14. Shaker Verlag, Aachen, Deutschland, 2012.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Aachener Informatik-Berichte, Software Engineering, Band 1. Shaker Verlag, März 2010.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. In *Workshop on Formal Methods for Open Object-based Distributed Systems*, IFIP Advances in Information and Communication Technology, pages 323–338. Chapmann & Hall, 1996.

[KRR14] Helmut Krcmar, Ralf Reussner, and Bernhard Rumpe. *Trusted Cloud Computing*. Springer, Schweiz, December 2014.

[KRS12] Stefan Kowalewski, Bernhard Rumpe, and Andre Stollenwerk. Cyber-Physical Systems - eine Herausforderung für die Automatisierungstechnik? In *Proceedings of Automation 2012, VDI Berichte 2012*, Seiten 113–116. VDI Verlag, 2012.

[KRV06] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Roles in Software Development using Domain Specific Modelling Languages. In *Domain-Specific Modeling Workshop (DSM'06)*, Technical Report TR-37, pages 150–158. Jyväskylä University, Finland, 2006.

[KRV07a] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Domain-Specific Modeling Workshop (DSM'07)*, Technical Reports TR-38. Jyväskylä University, Finland, 2007.

[KRV07b] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In *Conference on Model Driven Engineering Languages and Systems (MODELS'07)*, LNCS 4735, pages 286–300. Springer, 2007.

[KRV08] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: Modular Development of Textual Domain Specific Languages. In *Conference on Objects, Models, Components, Patterns (TOOLS-Europe'08)*, LNBIP 11, pages 297–315. Springer, 2008.

[KRV10] Holger Krahn, Bernhard Rumpe, and Stefen Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(5):353–372, September 2010.

[LRSS10] Tihamer Levendovszky, Bernhard Rumpe, Bernhard Schätz, and Jonathan Sprinkle. Model Evolution and Management. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 241–270. Springer, 2010.

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving Software Architecture Descriptions of Critical Systems. *IEEE Computer*, 43(5):42–48, May 2010.

[MRR10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. A Manifesto for Semantic Model Differencing. In *Proceedings Int. Workshop on Models and Evolution (ME'10)*, LNCS 6627, pages 194–203. Springer, 2010.

[MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. ADDiff: Semantic Differencing for Activity Diagrams. In *Conference on Foundations of Software Engineering (ESEC/FSE '11)*, pages 179–189. ACM, 2011.

[MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. An Operational Semantics for Activity Diagrams using SMV. Technical Report AIB-2011-07, RWTH Aachen University, Aachen, Germany, July 2011.

[MRR11c] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 592–607. Springer, 2011.

[MRR11d] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Modal Object Diagrams. In *Object-Oriented Programming Conference (ECOOP'11)*, LNCS 6813, pages 281–305. Springer, 2011.

[MRR11e] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Semantically Configurable Consistency Analysis for Class and Object Diagrams. In *Conference on Model Driven Engineering Languages and Systems (MODELS'11)*, LNCS 6981, pages 153–167. Springer, 2011.

[MRR13] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Synthesis of Component and Connector Models from Crosscutting Structural Views. In Meyer, B. and Baresi, L. and Mezini, M., editor, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 444–454. ACM New York, 2013.

[MRR14] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. Verifying Component and Connector Models against Crosscutting Structural Views. In *Software Engineering Conference (ICSE'14)*, pages 95–105. ACM, 2014.

[NPR13] Antonio Navarro Pérez and Bernhard Rumpe. Modeling Cloud Architectures as Interactive Systems. In *Model-Driven Engineering for High Performance and Cloud Computing Workshop*, CEUR Workshop Proceedings 1118, pages 15–24, 2013.

[PFR02] Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product Line Annotations with UML-F. In *Software Product Lines Conference (SPLC'02)*, LNCS 2379, pages 188–197. Springer, 2002.

[PR94] Barbara Paech and Bernhard Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *Proceedings of the Industrial Benefit of Formal Methods (FME'94)*, LNCS 873, pages 154–174. Springer, 1994.

[PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe-and-Filter Architectures. In *Congress on Formal Methods in the Development of Computing System (FM'99)*, LNCS 1708, pages 96–115. Springer, 1999.

[PR01] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kilov, H. and Baclavski, K., editor, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15*. Northeastern University, 2001.

[PR03] Jan Philipps and Bernhard Rumpe. Refactoring of Programs and Specifications. In Kilov, H. and Baclavski, K., editor, *Practical Foundations of Business and System Specifications*, pages 281–297. Kluwer Academic Publishers, 2003.

[Rin14] Jan Oliver Ringert. *Analysis and Synthesis of Interactive Component and Connector Systems*. Aachener Informatik-Berichte, Software Engineering, Band 19. Shaker Verlag, 2014.

[RK96] Bernhard Rumpe and Cornel Klein. Automata Describing Object Behavior. In B. Harvey and H. Kilov, editors, *Object-Oriented Behavioral Specifications*, pages 265–286. Kluwer Academic Publishers, 1996.

[RKB95] Bernhard Rumpe, Cornel Klein, and Manfred Broy. Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme - Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland, März 1995.

[RRRW15] Jan Oliver Ringert, Alexander Roth, Bernhard Rumpe, and Andreas Wortmann. Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. *Journal of Software Engineering for Robotics (JOSER)*, 6(1):33–57, 2015.

[RRW13a] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. From Software Architecture Structure and Behavior Modeling to Implementations of Cyber-Physical Systems. In *Software Engineering Workshopband (SE'13)*, LNI 215, pages 155–170, 2013.

[RRW13b] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. MontiArcAutomaton: Modeling Architecture and Behavior of Robotic Systems. In *Conference on Robotics and Automation (ICRA'13)*, pages 10–12. IEEE, 2013.

[RRW14] Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. *Architecture and Behavior Modeling of Cyber-Physical Systems with MontiArcAutomaton*. Aachener Informatik-Berichte, Software Engineering, Band 20. Shaker Verlag, December 2014.

[RSW+15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *Software Product Line Conference (SPLC'15)*, pages 141–150. ACM, 2015.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, München, Deutschland, 1996.

[Rum02] Bernhard Rumpe. Executable Modeling with UML - A Vision or a Nightmare? In T. Clark and J. Warmer, editors, *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, pages 697–701. Idea Group Publishing, London, 2002.

[Rum03] Bernhard Rumpe. Model-Based Testing of Object-Oriented Systems. In *Symposium on Formal Methods for Components and Objects (FMCO'02)*, LNCS 2852, pages 380–402. Springer, November 2003.

[Rum04] Bernhard Rumpe. Agile Modeling with the UML. In *Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF'02)*, LNCS 2941, pages 297–309. Springer, October 2004.

[Rum11] Bernhard Rumpe. *Modellierung mit UML, 2te Auflage*. Springer Berlin, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring, 2te Auflage*. Springer Berlin, Juni 2012.

[Rum16] Bernhard Rumpe. *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Aachener Informatik-Berichte, Software Engineering, Band 11. Shaker Verlag, 2012.

[SRVK10] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling: State of the Art and Research Challenges. In *Model-Based Engineering of Embedded Real-Time Systems Workshop (MBEERTS'10)*, LNCS 6100, pages 57–76. Springer, 2010.

[THR+13] Ulrike Thomas, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann. A New Skill Based Robot Programming Language Using UML/P Statecharts. In *Conference on Robotics and Automation (ICRA'13)*, pages 461–466. IEEE, 2013.

[Völ11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Aachener Informatik-Berichte, Software Engineering, Band 9. Shaker Verlag, 2011.

[Wei12] Ingo Weisemöller. *Generierung domänenspezifischer Transformationssprachen*. Aachener Informatik-Berichte, Software Engineering, Band 12. Shaker Verlag, 2012.

[ZPK+11] Massimiliano Zanin, David Perez, Dimitrios S Kolovos, Richard F Paige, Kumardev Chatterjee, Andreas Horst, and Bernhard Rumpe. On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In *Proceedings of the SESAR Innovation Days*. EUROCONTROL, 2011.